Títol: Sistema de deduplicació de CORDIS
Autor: Rubèn Arana Morera

Data: 12 de Juny de 2013

Director: Josep Lluís Larriba Pey
Departament del director: Arquitectura de Computadors (AC)

Titulació: Enginyeria Informàtica
Centre: Facultat d'Informàtica de Barcelona (FIB)
Universitat: Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

Facultat Informàtica de Barcelona
Universitat Politècnica de Catalunya

PROJECTE FINAL DE CARRERA

SISTEMA DE DEDUPLICACIÓ DE CORDIS

Alumne
Rubèn Arana Morera

Director
Josep Lluís Larriba Pey

Agraïments

M'agradaria expressar el meu agraïment a totes les persones que directa o indirectament m'han ajudat a tirar endavant aquest projecte.

En primer lloc a en Josep Lluís Larriba Pey, el director del projecte, per donar-me l'oportunitat d'incorporar-me a un grup com DAMA-UPC que no només m'ha permès realitzar el projecte de final de carrera, sinó també m'ha permès trobar el meu camí dins de la informàtica.

També a Raquel Pau, per orientar-me en la realització de tot el projecte i ensenyar-me tantes coses, a en Miguel Ángel Águila per estar disposat a ajudar en tot moment, i a la resta de persones del grup, tant als que encara hi són com als que han seguit altres camins, per obrir-me els braços des del primer moment.

Finalment, i no per això menys importants, m'agradaria donar les gràcies a totes aquelles persones, familiars i amics, que m'han donat suport i han confiat en mi en tot moment, fent-me veure sempre el costat positiu de les coses.

Índex

1 Motivació

Les eines informàtiques són un dels pilars on se sustenta la societat actual, també anomenada societat de la informació. Aquestes eines l'han fet evolucionar de tal manera que 50 anys enrere seria impensable en àmbits tan diversos com la medicina, l'educació o les relacions interpersonals. La difusió del coneixement i de la informació a nivell global, el tenir gairebé qualsevol resposta a un sol clic, és indubtablement una de les fites més importants que s'han assolit. Actualment tenim una gran quantitat d'informació disponible al nostre abast, de manera que el problema que se'ns planteja ja no és on trobar-la, sinó saber si aquesta informació és prou rigorosa, completa i de confiança. En tot això, tindran un paper molt important les bases de dades, que ens permeten emmagatzemar i treballar amb aquesta informació d'una manera ordenada i eficaç.

Un exemple més concret en aquest camp és la nova eina informàtica CORDIS(*Community Research and Development Information Service*), que és una plataforma que recull informació sobre les activitats europees d'investigació i desenvolupament (I+D). CORDIS és un portal públic en obert que permet la cerca de projectes europeus, *calls* i altres documents a partir de paraules clau i també filtrar les recerques segons múltiples criteris, com per exemple l'àrea temàtica. En la Figura 1.1 és pot veure un exemple de cerca de projectes a partir d'una paraula clau (en aquest cas, "database"). [1]

Aquesta funcionalitat permet facilitar informació a les diferents empreses o centres de recerca per tal d'obtenir finançament públic per la seva recerca mitjançant la sol·licitud d'un projecte europeu. Per tal de presentar un projecte europeu, és necessari complir una sèrie de condicions. Una d'aquestes condicions és la de realitzar el projecte juntament amb alguna altre institució, el que s'anomena un *partner*.

Com ja s'ha dit, CORDIS permet fer recerques sobre projectes, i des d'aquí es podria arribar a saber quines institucions col·laboren en un projecte determinat, però malauradament no està enfocat a la cerca de *partners*. Això fa que una institució que busqui

Figura 1.1: Cerca de projectes en la pàgina web de CORDIS.

la col·laboració d'una altra per tal d'optar a la realització d'un projecte europeu tingui moltes dificultats. Per tal de satisfer aquesta necessitat neix *Sciencea* [2].

Sciencea (provinent de *Science Analytics*) és un portal web del grup DAMA-UPC [3] que té com objectiu disposar de cerques intel·ligents que permetin no només la cerca de projectes europeus, sinó també de *partners* experts en una temàtica. D'aquesta manera es facilita el poder establir una possible comunicació entre dues institucions que acabin per formalitzar un acord de cooperació per tal de presentar un projecte europeu de forma conjunta. En la Figura 1.2 es pot veure una vista general de *Sciencia*.

Tot i que *Sciencea* pretén ser un cercador de *partners*, no és només això. Aquest portal també vol ser un referent en informació relacionada amb projectes europeus, i és per això que busca donar un valor afegit a les dades. Algunes d'aquestes funcionalitats que donen un valor afegit permeten veure:

- Els temes sobre els quals es realitzen més projectes a nivell europeu (Figura 1.3a).

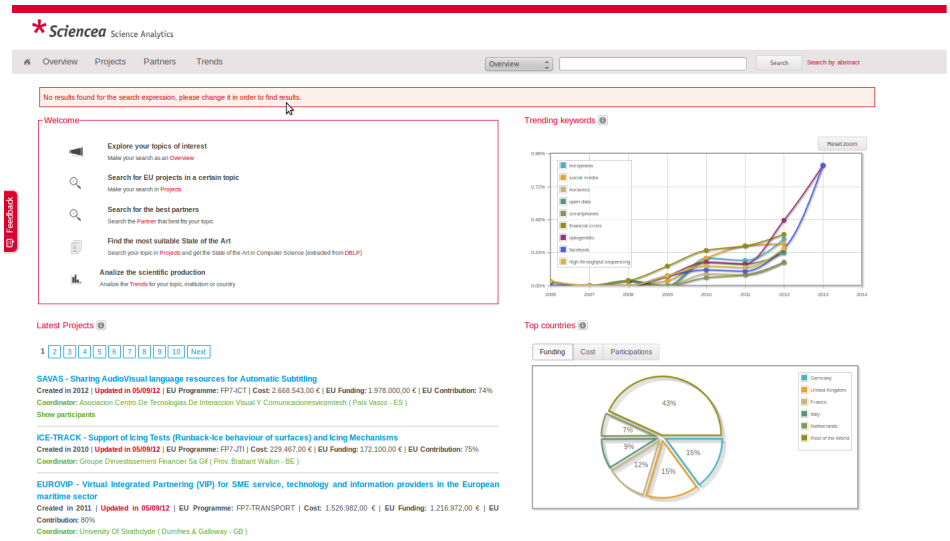
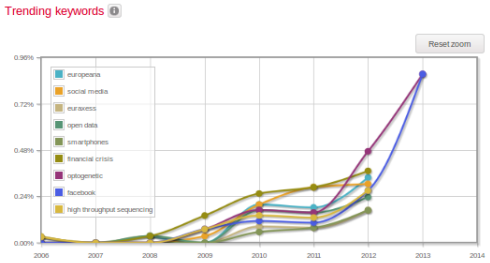


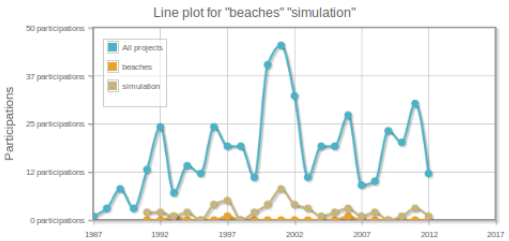
Figura 1.2: Visió general de la pàgina web de *Sciencea*.

- L'evolució de les participacions d'una institució, és a dir, l'evolució del nombre de projectes en els que participa una institució al llarg dels anys (Figura 1.3b).
- Informacions referents al finançament de projectes per països i segons la temàtica del projecte (Figura 1.3c).
- La distribució de països on es fan projectes sobre un tema concret (Figura 1.3d).
- Els projectes o institucions més similars a un projecte o institució donada (Figura 1.3e).
- Els col·laboradors o *partners* a partir d'una institució (Figura 1.3f).

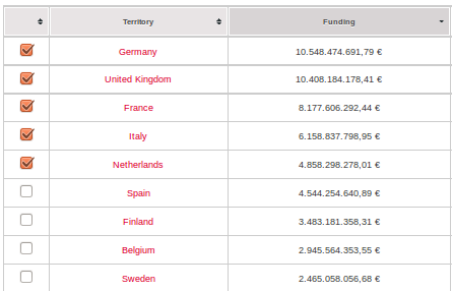
Totes aquestes informacions, provenen de les dades a les que CORDIS permet accedir en la seva pàgina web. No obstant, aquestes dades no estan en tan bon estat com seria desitjable. Segons el projecte, hi ha dades incorrectes, redundants, incompletes... i encara s'agreuja més en el cas de les institucions, ja que tot i tenir un identificador propi, aquest no és de confiança. Això es tradueix en múltiples casos de projectes i institucions repetides, que, de no ser tractats, comporten de manera automàtica molt poca fiabilitat a totes les funcionalitats ofertes per *Sciencea*.



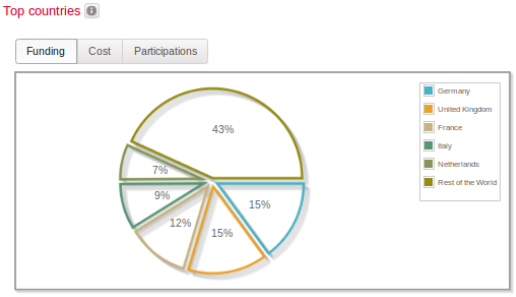
(a) Trending Keywords



(b) Evolució participacions

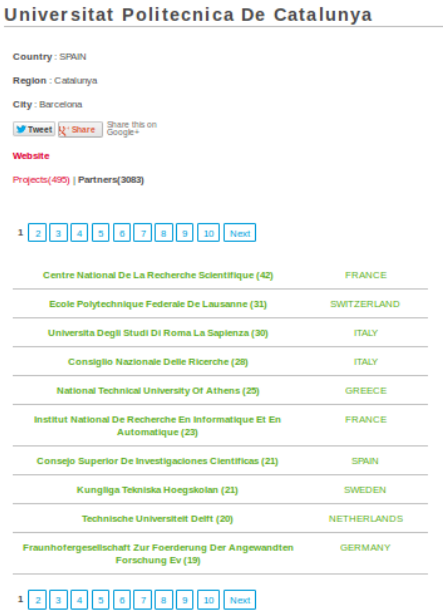


(c) Funding



(d) Distribucció de projectes

(e) Institucions similars



(f) Partners

Figura 1.3: Features de Sciencea.

Num	Nom	País	Ciutat	Adreça
1	Universitat Politècnica de Catalunya	-	Barcelona	Jodri Girona, 31 (Espanya)
2	Universidad Politécnica de Cataluña	ES	Barcellona	-
3	Techincal University of Catalonia - UPC	ES	Barcelona (Catalunya)	Carrer de Jordi Girona, 31, 08034 Barcelona
4	UPC	-	-	C/ Jordi Girona, Barcelona, España
5	Facultat d'Informàtica de Barcelona (UPC)	ES	Barsalona	-

Figura 1.4: Exemple de dades extretes directament de CORDIS.

Un exemple d'això es pot veure en el cas de les aproximadament 500 participacions de la Universitat Politècnica de Catalunya en diferents projectes. Dins d'aquestes participacions, podem trobar registres que contenen, entre altres dades, les que s'exemplifiquen en la Figura 1.4 . En aquesta taula es pot observar com, per al que hauria de ser una mateixa institució, hi ha entrades molt diferents entre sí.

Es pot observar que hi ha 5 noms diferents per a referir-se a la mateixa institució, la qual cosa es força habitual en les dades. Es poden trobar de manera recurrent casos de noms diferents degut a diverses causes. Algunes de les més comunes son:

- Canvis d'idioma (1, 2 i 3 en la taula) que poden comportar petits canvis (alguns caràcters diferents) o convertir-se en un nom totalment diferent com en 3.
- Nom de la institució com a sigla (4).
- Aparició d'una part de la institució (un departament, una facultat...) en el nom (5).

D'altra banda, pot ser que falti informació referent al país (1,4), a la ciutat (4) o a l'adreça (2,5). Però no només això, sinó que les ciutats poden tenir errors tipogràfics (2,5) o anar acompanyades de una divisió geogràfica superior (3). A més, també ens podem trobar molts formats d'adreça diferents (a més de noms en diferents idiomes, o errors tipogràfics), complicant encara més aquesta tasca.

En general, solucionar aquest tipus de problemes de duplicació manualment és inviable, i més quan es treballa amb grans volums de dades com és el cas de CORDIS on es tenen més de 400.000 institucions abans de ser deduplicades. Això és degut a que realitzar una comparació de tots contra tots és un problema quadràtic ($O(N^2)$, on N és el nombre de registres a comparar) i per tant molt costós de solucionar.

Per tal de millorar el cost d'aquest procés, apareixen les tècniques de *Record Linkage*, que tenen com a finalitat aparellar aquelles entrades que tenen un cert grau de semblança de forma automàtica. Això permetrà netejar una base de dades de les seves entrades repetides i fusionar o creuar dues bases de dades sense quedar-se amb duplicats.

Dins del grup DAMA-UPC ja s'ha treballat anteriorment en tot el tema relacionat amb el *Record Linkage* i les deduplicacions. Concretament es disposa d'un software per trobar duplicats en un conjunt de dades d'una manera eficient. Aquest software s'anomena *Daurum*. Avui dia, aquesta tecnologia restringeix les possibles maneres d'agrupació i comparació dels registres. Per tant, tot i complir la seva funció, aquestes limitacions impliquen en algunes situacions haver de realitzar una revisió final que pot ser molt feixuga al tractar grans volums de dades. Aquesta és una de les raons per les quals es vol treballar de cara a obtenir un sistema més automatitzat i parametritzable de deduplicació.

L'objectiu d'aquest projecte és construir un *framework* de deduplicació aplicable a un sistema que permeti deduplicar les dades de CORDIS, la qual cosa suposa realitzar un tractament molt precís i minuciós de les dades. Aquest tractament haurà de fer de manera completament automàtica degut al gran volum d'informació que tenim entre mans i suposarà fer de *Sciencea* un producte final que sigui un referent en l'àmbit de la recerca a nivell europeu. A més, aquest *framework* ha de poder-se reutilitzar en altres projectes de deduplicació del grup, de manera que s'ha de realitzar de la manera més genèrica possible.

2 Conceptes previs

Tal i com s'ha explicat en el capítol anterior, l'objectiu d'aquest projecte és realitzar una deduplicació de les dades de CORDIS. Aquest objectiu concorda amb un dels escenaris on s'utilitza el *Record Linkage* anomenat *Object Integration*. Aquest procés consisteix en intentar establir relacions entre diferents objectes que tenen alguna similitud, ja sigui en les seves propietats o en el seu comportament. Tot i així, degut a que les dades a partir de les quals es treballarà presenten freqüents problemes de *dades* incompletes, invàlides o mancants, prèviament s'haurà de passar per un altre procés dins del *Record Linkage* com és el de *Data Cleansing*. Aquest és un procés consistent en la detecció i eliminació d'errors i inconsistències en les dades per tal de millorar-ne la seva qualitat.

2.1 Record Linkage

2.1.1 Contextualització

La idea de *Record Linkage* neix a la dècada dels quaranta. Poc després es van publicar els fonaments probabilístics de la teoria moderna, i posteriorment ja es va formalitzar un model matemàtic que segueix sent vàlid avui en dia.

Situem el *Record Linkage* dins del camp de la gestió de la informació, més concretament dins de les tècniques de re-identificació que pretenen establir relacions entre les diferents entitats que hi ha en diverses fonts de dades. A part dels dos processos que s'utilitzaran al llarg del projecte (*Data Cleansing* [4] i *Object Integration* [5]), el fet de trobar aquestes relacions té sentit en altres escenaris:

- *Schema Matching*: donats dos esquemes diferents, trobar les relacions que hi pot haver entre els seus atributs. [6]

- *Data Integration*: donades diverses fonts de dades que semblen incompatibles (degut a la gran varietat de formes que poden adoptar els conjunts de dades), crear una vista integrada de totes aquestes fonts. [7]

- *Master Data Management*: té com a principal objectiu la creació de diferents processos que permetin garantir el control de les dades i poder-ne fer un ús controlat. Per aconseguir-ho, es requerirà recollir, agregar, creuar, consolidar i assegurar la qualitat de la informació.

2.1.2 Procés general de Record Linkage

El *Record Linkage* consta de diferents etapes [8] que s'executen de forma seqüencial fins a arribar a l'objectiu final d'arribar a un conjunt de similituds que són acceptades. Les etapes dins del procés de *Record Linkage* son les següents:

- Fase de Preprocés: És la primera fase del procés. S'encarrega de normalitzar la informació de les diferents fonts de dades, netejar-la i unificar-la per tal de poder treballar amb ella d'una manera més còmoda.

- Fase de Comparació: Aquesta fase s'encarrega de comparar tots els registres entre ells. S'utilitza una funció de comparació que servirà per obtenir el grau de similitud o pes entre dos registres comparats. Aquest pes s'obtindrà a partir de la suma (que pot ser ponderada) dels graus de similitud entre els seus atributs comparats entre ells. Al final d'aquesta fase s'obtindrà un llistat de parelles de registres juntament amb el pes calculat. Aquesta és la fase més costosa, ja que té un cost quadràtic respecte el nombre de registres ($O(N^2)$, on N és el nombre de registres) en el cas de voler comparar tots els registres contra tots . És per això, que es busquen alternatives com els mètodes de *blocking* per tal d'intentar reduir-ne la complexitat.

- Fase d'Avaluació: Fase final del procés. A partir de les parelles de registres i tenint en compte la seva similitud, es decideix si dos registres són el mateix o no. Aquesta fase es pot realitzar de forma manual, analitzant les parelles una a una o també de forma automàtica, acceptant com a iguals aquelles parelles que tinguin un pes superior a un llindar o *threshold*.

2.1.3 Mètodes de blocking

Com ja s'ha comentat, la comparació de cada registre amb tots els altres és molt costosa a nivell computacional, de manera que s'han de buscar alternatives, sobretot quan es parla de grans volums de dades.

Els mètodes de *blocking* divideixen el conjunt de dades en diferents blocs basats en els atributs dels registres. D'aquesta manera, cada un dels registres anirà a parar a un bloc i només es compararà amb els altres registres que formen part del bloc, estalviant així moltes comparacions. Els mètodes clàssics de *blocking* son el *Standard Blocking* i el *Sliding Window*.

El problema més important amb el que ens podem trobar al fer ús dels diferents mètodes de blocking és que si dos registres són duplicats, però per alguna raó no estan en el mateix bloc, serà impossible detectar-los com a una parella de duplicats. És per això que cal ser molt curosos a l'hore d'escollir el criteri pel qual es generaran els diferents blocs.

Standard Blocking

El mètode de *Standard Blocking* agrupa els registres en blocs a partir d'una clau de bloc, la qual es defineix a partir dels atributs del registres. Per exemple es podria agafar com a clau l'atribut *Ciutat*. D'aquesta manera, només es realitzarien les comparacions entre 2 registres de la mateixa ciutat.

Bloc	Nom	País	Ciutat	Adreça
Bloc 1	Universitat Politècnica de Catalunya	-	Barcelona	Jordi Girona, 31 (Barcelona)
Bloc 1	Universitat Politècnica de Catalunya	-	Manresa	Av. de les Bases, 61 08242 Manresa
Bloc 1	Universitat Politècnica de Catalunya	ES	Vilanova i la Geltrú	-
Bloc 1	Universitat de Barcelona	-	Barcelona	Gran Via, 585 (Barcelona)
Bloc 1	Universitat Pompeu Fabra	ES	Barcelona	-

Figura 2.1: Exemple de registres que es compararien tots contra tots en un sistema de *Record Linkage* convencional.

Com es pot veure en l'exemple de la Figura 2.1 , en un sistema de *Record Linkage* convencional, tots els registres es compararien contra tots. En aquest cas concret, comportaria realitzar 10 comparacions.

Per contra, si s'utilitza el mètode de *Standard Blocking*, en aquest cas concret només serien necessàries 3 comparacions, com mostra la Figura 2.2 . És clar que el nombre de comparacions es redueix d'una manera dràstica. Si abans la fase de comparació tenia un cost quadràtic, ara el cost ve determinat per la funció $O(B \cdot N)$, on B és la mida del bloc que conté més registres i N és el nombre de blocs.

Bloc	Nom	País	Ciutat	Adreça
Bloc 1	Universitat Politècnica de Catalunya	-	Barcelona	Jordi Girona, 31 (Barcelona)
Bloc 1	Universitat de Barcelona	-	Barcelona	Gran Via, 585 (Barcelona)
Bloc 1	Universitat Pompeu Fabra	ES	Barcelona	-
Bloc 2	Universitat Politècnica de Catalunya	-	Manresa	Av. de les Bases, 61 08242 Manresa
Bloc 3	Universitat Politècnica de Catalunya	ES	Vilanova i la Geltrú	-

Figura 2.2: Exemple de registres separats per *Standard Blocking*.

És important escollir com a clau un atribut que considerem que és menys propens a tenir errors [9], ja que si no dos registres, que en realitat són el mateix, podrien quedar ubicats en blocs diferents, i, al no realitzar la comparació entre ells, no detectaríem aquesta similitud. Això es pot solucionar realitzant varies iteracions d'aquest procés, escollint a cada una d'elles una clau de bloc diferent, de manera que es generaran noves similituds. Si es realitzen aquestes iteracions, apareixerà un nou problema: s'estaran generant similituds repetides. Així doncs, s'haurà d'incorporar al procés una nova fase que ens permeti eliminar similituds repetides.

Sliding Window

El mètode de *Sliding Window* parteix de registres que estan ordenats segons una clau d'ordenació que pot estar composada per un o més atributs. Un cop es tenen els registres ordenats, s'ha de definir la mida de la finestra o *window*. Aquesta mida serà el nombre màxim de comparacions que es realitzaran amb un sol registre.

Un aspecte important a tenir en compte és definir correctament el valor de la mida de la finestra, ja que si és massa petit no es compararan registres que podrien ser la mateixa

entitat. D'altra banda, tampoc pot ser massa gran, ja que si no es realitzarien moltes comparacions inútils degut a que els registres estan ubicats en blocs diferents. [10]

Així doncs, el que es farà serà comparar el primer dels registres amb els w-1 següents, on w es és la mida de la finestra. A continuació es compararà el segon registre amb els w següents, i així fins a arribar a l'últim registre. En la Figura 2.3 es pot veure el funcionament del mètode de *Sliding Window*.

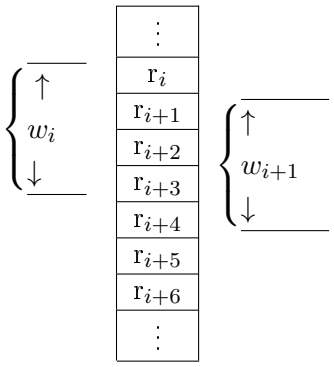


Figura 2.3: Il·lustració del desplaçament de les finestres en el mètode de *Sliding Window* amb una finestra $w = 4$.

3 Framework de deduplicació: DEXDUP

En aquest tercer capítol, s'explicarà el funcionament del *framework* de deduplicació utilitzat per a realitzar *Record Linkage* durant la càrrega de les dades. En concret, s'utilitzarà per deduplicar ciutats, projectes i institucions.

3.1 Descripció

Dexdup és un *framework* programat en Java per tal de facilitar les tasques de deduplicació, tant sobre fitxers com sobre diferents tipus de bases de dades. En el cas concret d'aquesta deduplicació s'aplicarà a fitxers de tipus *csv*, ja que volem que la càrrega de les dades a la base de dades es produeixi un cop tota la informació ja estigui tractada i deduplicada. *Dexdup* posa a disposició del programador una estructura basada en interfícies que el programador haurà d'implementar, adaptant-les a les seves necessitats concretes. Tot i així, és el nucli de Dexdup el que controla el fil de l'execució, utilitzant la implementació de les interfícies especificada a través d'un fitxer de configuració en *xml*. L'esquema de Dexdup és el mostrat en la Figura3.1.

La classe on està el *main()* és *Dexdup*. Des d'aquesta classe principal sempre s'obtindrà la informació de la configuració introduida mitjnçant un fitxer *xml* a través del *XMLConfigurationProvider*, una implementació del *ConfigurationProvider*. Una vegada s'hagi obtingut aquesta informació de configuració, es cridarà a *DeduplicationInvocation*, que carregarà aquesta informació, crearà les classes necessàries i executarà tot el procés de deduplicació pròpiament dit, que s'explicarà amb detall més endavant.

Des del *XMLConfigurationProvider*, s'executarà el *DedupConfigHandler*, que mitjançant *SAX*, parsejarà el fitxer de configuració *xml* i crearà una o varies instàncies de *DeduplicationConfiguration*. Aquesta classe contindrà la informació específica de configuració dels

preprocessos i dels postprocessos, així com del *BlockProvider* i del *ComparisonMethod*, tots ells components de Dexdup.

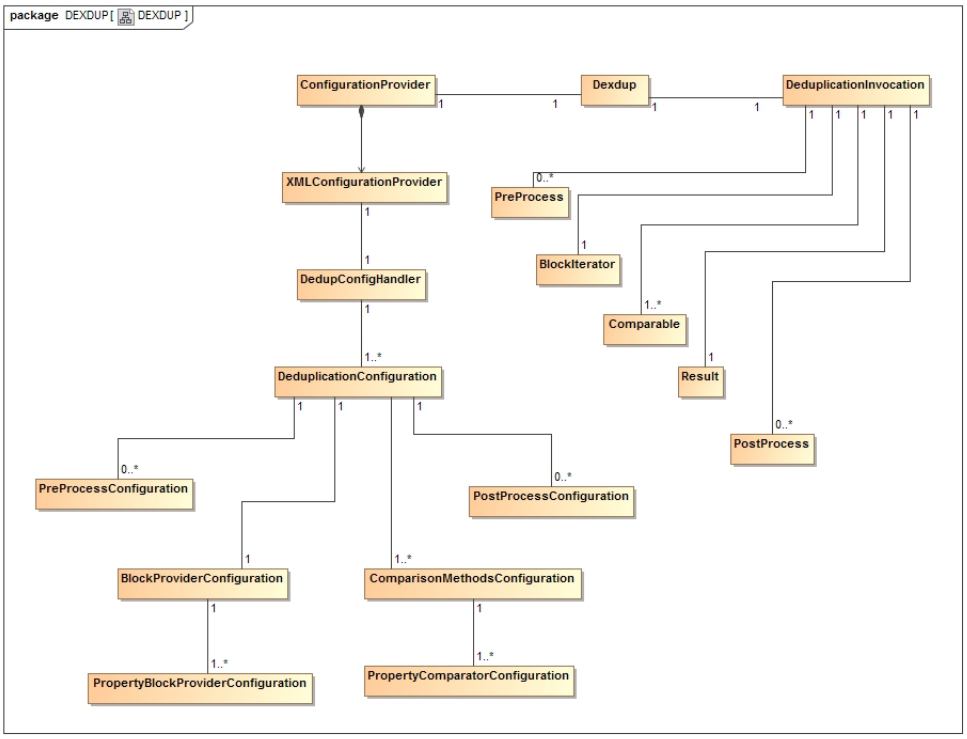


Figura 3.1: Diagrama de classes de *Dexdup*

Així doncs, aquest *framework* que permet executar deduplicacions disposa de diferents tipus de components tal i com s'ha vist:

- *PreProcess*: conjunt de processos a realitzar abans de la deduplicació.
- *BlockProvider*: el proveïdor o conjunt de proveidors de blocs de registres a comparar.
- *ComparisonMethod*: els propis comparadors.
- *Result*: el resultat de la deduplicació.
- *PostProcesss*: conjunt de processos a realitzar un cop acabada la deduplicació.

3.2 Components principals

3.2.1 BlockProvider

Com s'ha dit anteriorment, per tal de realitzar *Record Linkage*, les dades s'organitzen en blocs d'entrades per tal d'evitar haver fer comparacions tots contra tots (amb cost $O(n^2)$). El *BlockProvider* serà l'encarregat de decidir quins seran i com es faran els blocs amb les entrades a comparar.

Això ho realitzarà a partir de la interfície anomenada *BlockIterator*, que serà una de les interfícies que haurà de ser modificada pel programador. La classe *BlockIterator* s'implementa com un iterador que te les funcions *hasNext()* i *next()*, on el resultat de realitzar una crida a *next()* és l'estructura de dades que implementa el bloc. En aquesta estructura de dades, que serà una llista, es compararà el primer objecte contra tota la resta, seguint la metodologia *One-Vs-All*.

Objecte1
Objecte2
Objecte3
Objecte4

Figura 3.2: Exemple de bloc

Aquest bloc d'exemple de la Figura 3.2 generaria les comparacions *Objecte1 vs Objecte2*, *Objecte1 vs Objecte3* i *Objecte1 vs Objecte4*.

A més, la interfície *BlockIterator* defineix una funció *setBlockSize()* per tal de indicar la mida màxima del bloc. Hi ha disponible la classe *DefaultBlockIterator* que implementa la funció *setBlockSize*() com a *setter* de l'atribut *blockSize*.

D'altra banda, també existeix l'opció d'utilitzar diferents *BlockIterators* a la vegada per tal d'aconseguir parelles diferents a comparar. Això serà possible al fer us de la classe *CompositeBlockIterator*, que executarà els diferents *BlockIterators* de forma seqüencial. Això és especialment útil si es volen comparar unes mateixes dades agrupades segons diferents criteris.

3.2.2 ComparisonMethod

Per tal de fer les comparacions entre les parelles d'entrades que hem escollit en el *BlockIterator*, necessitem comparadors. En el context del *framework*, un comparador és una implementació de la classe *Comparable* o *CompositeComparable*.

Aquestes noves classes hauran d'implementar una única funció *compare()* que rep dos Object i retorna un *double*, indicant el nivell de similitud entre aquests dos objectes. La diferencia entre les dos interfícies es la següent:

- *Comparable*: és la classe més bàsica que únicament implementa l'operació esmentada.

- *CompositeComparable*: a més a més de la funció *compare()* té una nova funció *setComparatorsMap()*. Aquesta nova funció, que té com a argument un Map amb diferents *Comparables*, es crida al instanciar la classe. El programador podrà fer servir tots aquests *Comparables* en el seu codi, i seran especificats en l'arxiu de configuració. D'aquesta manera, es podran utilitzar diferents comparadors en una mateixa deduplicació. Aquests retornaran resultats diferents que seran tractats per la classe *Result* per tal d'obtenir una similitud final entre els dos objectes.

A més, també es disposa d'una classe abstracta amb el nom de *DefaultCompositeComparable*. Aquesta nova classe implementa la funció *setComparatorsMap()* i ja te un Map definit com un atribut privat per tal de facilitar la implementació del comparador.

Cal dir que una configuració de deduplicació pot contenir més d'un comparador. En aquest cas, tots ells seran executats de manera seqüencial, amb l'ordre definit al fitxer de configuració.

Una altre peculiaritat d'aquest *framework* és que durant la pròpia fase de comparació permet prendre decisions sobre l'estat de la mateixa, sempre i quan s'estengui el comparador de la interfície *DecisionAware*. D'aquesta manera es podrà obtenir en quin estat està la comparació (si acceptat, rebutjat o desconegut), juntament amb un missatge informant de la raó d'aquest estat. Això permetrà acceptar o rebutjar una parella d'entrades basant-se no només en les dues entrades sinó també amb altres factors, com per exemple si ja s'ha acceptat com a duplicat d'una de les entrades anteriorment.

3.2.3 Result

La implementació de la interfície *Result* rep i processa el resultat de cada una de les diferents comparacions. Aquests resultats es poden tractar de la manera que es vulgui per tal d'obtenir un resultat final de similitud. Segons el cas i segons com estiguin implementats els comparadors, un bon resultat final podria ser la mitja aritmètica dels resultats de cada un dels comparadors, el màxim o el mínim.

Després de cada comparació s'executa la funció *addResult()*, que serà l'encarregada de decidir si s'accepta o es rebutja una comparació basant-se principalment en si la similitud entre les dues entrades supera un valor llindar. Aquest valor llindar podrà ser parametritzable a partir del fitxer de configuració en *xml*. Tot i així, si s'extén de la interfície *DecisionAwareResult*, també es podrà fer ús de la informació de l'estat de la comparació i del motiu d'aquest estat per tal de prendre una decisió sobre si dues entrades es consideren una parella de duplicats o no.

Per tal que la implementació de *Result* detecti quin comparador s'ha fet servir, abans d'afegir un resultat amb *addResult()* es crida a *setComparisonConfiguration(),* on es passa un objecte que conte la configuració del *Comparable* o *CompositeComparable* especificada al *xml* de configuració.

En el cas que es necessiti realitzar alguna tasca en el moment en el que totes les comparacions ja han finalitzat, sempre hi haurà l'opció d'implementar la funció *execute()*, que sempre s'executa en aquest precís moment.

3.3 Diagrama de seqüència

En la Figura 3.3 podem veure el diagrama de seqüència del procés que realitza *Dexdup*.

El funcionament serà el següent: es preguntarà al *BlockProvider* si hi ha més blocs a comparar. Si n'hi ha, es retornarà el proper bloc trobat des del *BlockIterator* implementat pel programador. Per a totes les parelles a comparar del bloc, es realitzarà el càlcul de similitud per als comparadors desitjats, retornan per a cada comparació un valor *similarity*. Aquesta similitud serà el valor que arribarà a la classe Result. Aquesta classe serà la que tractarà el resultat i la que decidirà si la parella és un duplicat o no i segons

això, actuarà en conseqüència, escrivint la parella de duplicats en un fitxer de duplicats acceptats o rebutjats, per exemple.

Quan ja no hi hagi més blocs a retornar per a un *BlockProvider* determinat, és mirarà si n'hi ha un altre. En aquest cas, el procés s'aniria repetint fins a realitzar les comparacions dels blocs retornats per tots els *BlockProviders* existents en el fitxer de configuració.

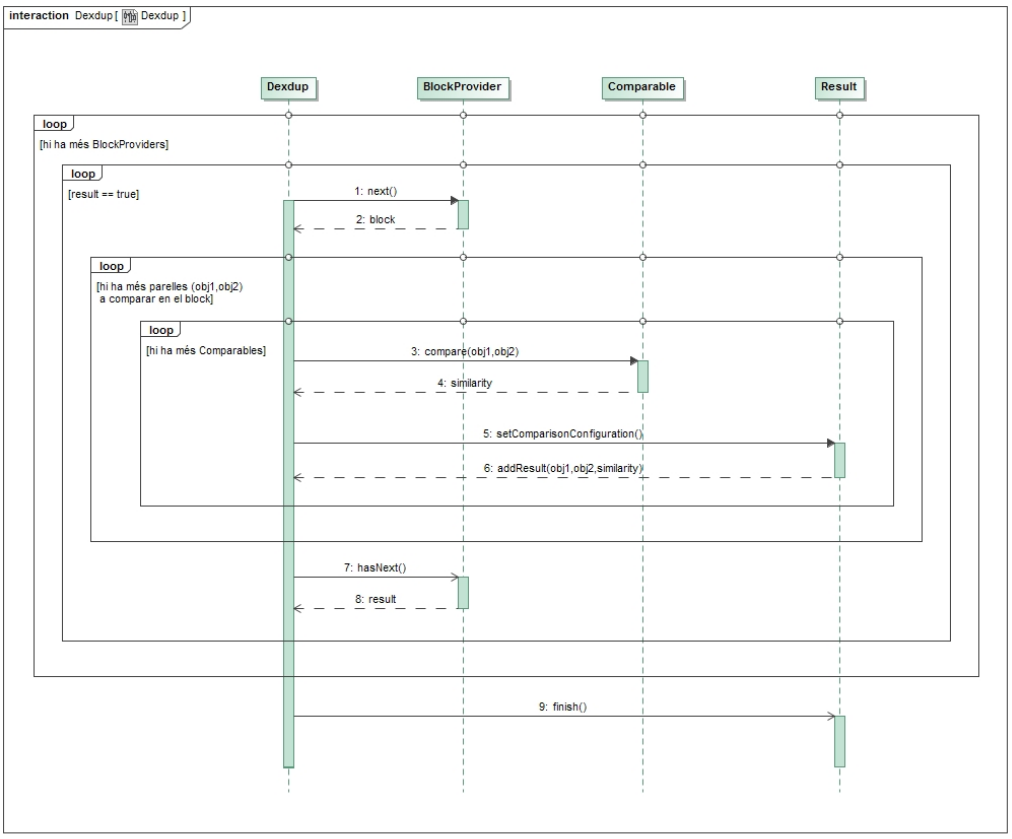


Figura 3.3: Diagrama de seqüència de *Dexdup*.

3.4 L'arxiu de configuració

Per configurar el *framework* de *Dexdup* cal crear un arxiu *xml* anomenat *dexdup.xml*. En aquest arxiu s'indica quines són les classes que implementen les interfícies del *framework,*

així com altres atributs i propietats.

L'arxiu té com arrel l'etiqueta *config*.

```
<config>
```

Tot seguit s'ha d'indicar la configuració d'una o més deduplicacions:

```
<deduplication name="dedup" active="true" avoidDuplicatedPairs="true">
```

- *name* és un atribut únic que identifica la deduplicació.
- *active* és un booleà que marca si aquesta deduplicació s'ha d'executar en cas que es demani que s'executin totes les deduplicacions.
- *avoidDuplicatedPairs* és un booleà que indica si volem deixar de comparar 2 entrades que ja han estat comparades anteriorment. El que es realitza, bàsicament és la creació d'un llistat en memòria de totes les parelles ja comparades. Això pot causar problemes de memòria si es realitzen un gran nombre de comparacions, però millora la eficiència en conjunts de dades petits. En el cas particular d'aquest projecte, les comparacions a realitzar són massa nombroses com per utilitzar aquesta opció i no tenir problemes amb una màquina estàndard.

A continuació s'indica la configuració del *Block Provider*:

```
<blockProvider bean="BlockIteratorImpl" blockSize="100">
```

- *bean* és el nom de la classe que implementa el *BlockIterator*.
- *blockSize* és un atribut opcional que indica la mida màxima del bloc. En cas que es defineixi l'atribut *blockSize* es farà una crida a la funció *setBlockSize()* de la classe que implementa el *BlockIterator*.

En aquest punt es defineixen un o més *ComparisonMethod*:

```
<comparisonMethod name="comp1" bean="ComparableImpl">
```

- *bean* és el nom de la classe que implementa el *Comparable*.
- *name* és l'identificador únic del *comparisonMethod*.

En cas que la classe indicada al *comparisonMethod* sigui un *CompositeComparable*, s'han de especificar quins comparadors es volen fer servir amb la etiqueta *propertyComparator*:

```
<comparisonMethod name="compMethod1" bean="CompositeComparableImpl">
  <property name="operator" value="max.o.min.o.avg"/>
  <propertyComparator name="compa1" comparator="ComparableImpl1"/>
  ...
  <propertyComparator name="compN" comparator="ComparableImplN"/>
</comparisonMethod>
```

- *name* és el nom que tindrà aquest comparador en el Map disponible a la implementació del *CompositeComparable*.
- *comparator* és la classe que implementa el *Comparable*.

A més, *comparisonMethod* pot tenir una *property* amb el nom de *operator*, que indiqui si es vol agafar el resultat màxim, mínim o mitjà de totes les comparacions segons es posi en el seu *value* "*max*", "*min*" o "*avg*" respectivament.

El *framework* disposa d'una bateria de *Comparables* ja implementats per tal de ser utilitzats en aquest context, que permeten operacions bàsiques de comparació de *strings* per exemple.

El següent punt a definir és la classe que implementa *Result*:

```
<result bean="ResultImpl">
```

- *bean* és la classe que implementa *Result*.

Aquests són els elements bàsics necessaris per a una configuració minimal del *framework*.

Després d'aquesta configuració, es poden afegir un número il·limitat de postprocessos. Un postprocés d'una implementació de la classe *PostProcess* és una interfície amb un única funció *execute()* sense cap argument.

```
<postProcess bean="PostProcessImpl">
```

- *bean* és la classe que implementa *PostProcess*.

A més, tots els elements de la configuració accepten contenir una o més etiquetes *property* per tal d'indicar a les implementacions realitzades de les classes bàsiques certs valors necessaris pel seu correcte funcionament. Aquests valors podrien anar des del valor del llindar per acceptar o rebutjar una parella de duplicats, el nom del fitxer on es volen escriure els resultats obtinguts o el nom d'una base de dades temporal utilitzada a l'hora de fer comparacions.

```
<property name="propName1" value="propValue1"/>
```

- *name* és el nom de la propietat.
- *value* és el valor de la propietat.

Quan es defineix una *property* a un element, es farà una crida a una funció que depèn del nom de la mateixa. Com a exemple:

```
<result bean="ResultImpl">
  <property name="outputFile" value="result.csv"/>
</result>
```

D'aquesta manera es cridaria a la funció *setOutputFile()* al instanciar la classe indicada al *bean* passant *value* com a argument.

El *framework* detecta automàticament el tipus de l'argument segons la definició de la funció. No obstant, els arguments han de ser objectes Java (int → Integer, double → Double).

Al *value* d'una *property* es poden posar elements de tipus String, Integer, Double, Float, Boolean i llistes dels anteriors, sempre que es faci entre claudàtors, separats per comes i sense espais. (e.g. "[text1,text2,text3]").

Exemple complert

Per claredat, s'adjunta una configuració d'exemple completa:

```
<config>

  <deduplication name="dedup" active="true" avoidDuplicatedPairs="true">

    <blockProvider bean="BlockIteratorImpl" blockSize="100">
      <property name="inputFile" value="input"/>
    </blockProvider>

    <comparisonMethod name="compMethod" bean="CompositeComparableImpl">
      <property name="myInt" value="1"/>
      <propertyComparator name="comp1" comparator="ComparableImpl"/>
    </comparisonMethod>

    <result bean="ResultImpl">
      <property name="outputFile" value="output"/>
    </result>

    <postProcess bean="PostProcessImpl">
      <property name="myIntegerList" value="[1,2,3]"/>
    </postProcess>

  </deduplication>

</config>
```

3.5 Exemple de deduplicació de noms de persona

Per tal d'aclarir el fucionament del framework de Dexdup, a continuació es realitzarà un exemple senzill de deduplicació de noms de persona. El nostre conjunt de dades serà els 5 noms mostrats a la Figura 3.4 .

Nom	Cognom1	Cognom2
Maria	Claramunt	Nogués
Lluís	Claramunt	Noges
Pere	Solsona	Sabater
Josep Lluís	Claramunt	Nogués
Maria del Mar	Claramont	Noguès

Figura 3.4: Conjunt de dades inicial.

El primer pas serà aplicar els possibles preprocessos que es vulguin realitzar. Un preprocés habitual en aquest tipus és el de l'ordenació de les dades. Així doncs, aquest fitxer es podria ordenar alfabèticament pel cognom1, i en cas d'igualtat, decidir segons l'ordre alfabètic segons el nom. D'aquesta manera, el fitxer de dades abans de començar el procés de deduplicació seria el mostrat en la Figura 3.5.

Nom	Cognom1	Cognom2
Josep Lluís	Claramunt	Nogués
Lluís	Claramunt	Noges
Maria	Claramunt	Nogués
Maria del Mar	Claramont	Noguès
Pere	Solsona	Sabater

Figura 3.5: Conjunt de dades ordenat.

Un cop ordenat, el *BlockIterator* és el que proporcionaria els blocks de persones a les que comparar. En aquest cas, es farà servir el mètode de *Sliding Window* amb un tamany de block de 3, és a dir, el *BlockIterator* retornarà conjunts de com a molt 3 persones on la primera es compararà amb les altres dues. D'aquesta manera els blocks que es retornaran seran els mostrats en la Figura 3.6.

Tenin en compte aquests blocks, es realitzaran un total de 7 comparacions (el primer de cada block contra les altres persones de cada block). Aquestes comparacions es podrien realitzar de moltes formes diferents. Per l'exemple en qüestió, se suposarà que s'utilitza

Nom	Cognom1	Cognom2
Josep Lluís	Claramunt	Nogués
Lluís	Claramunt	Noges
Maria	Claramunt	Nogués

Nom	Cognom1	Cognom2
Lluís	Claramunt	Noges
Maria	Claramunt	Nogués
Maria del Mar	Claramont	Noguès

Nom	Cognom1	Cognom2
Maria	Claramunt	Nogués
Maria del Mar	Claramont	Noguès
Pere	Solsona	Sabater

Nom	Cognom1	Cognom2
Maria del Mar	Claramont	Noguès
Pere	Solsona	Sabater

Figura 3.6: Blocks retornats.

algun tipus de comparador de Strings (ja es veurà més endavant en el capítol referent a la deduplicació de regions alguna possibilitat).

Així doncs, obtenim per cada una de les comparacions un valor de similitud:

- "Josep Lluís Claramunt Nogués" vs "Lluís Claramunt Noges": 0.71
- "Josep Lluís Claramunt Nogués" vs "Maria Claramont Noguès": 0.53
- "Lluís Claramunt Noges vs "Maria Claramunt Noguès": 0.63
- "Lluís Claramunt Noges vs "Maria del Mar Claramont Noguès": 0.50
- "Maria Claramunt Noguès" vs "Maria del Mar Claramont Noguès": 0.73
- "Maria Claramunt Noguès" vs "Pere Solsona Sabater": 0.18
- "Maria del Mar Claramont Noguès" vs "Pere Solsona Sabater": 0.23

Per a cada una d'aquestes comparacions, es cridarà a la funció *addResult,* on es passaran els dos noms i el valor de la similitud de la comparació. D'aquesta manera, la classe *Result* el que farà és mirar si aquest valor de similitud és major o menor que un llindar donat, que per exemple se suposarà de 0.7. Així doncs, les parelles que superen aquest valor, i que per tant es consideraran la mateixa persona seran:

- "Josep Lluís Claramunt Nogués" - "Lluís Claramunt Noges"
- "Maria Claramunt Noguès" - "Maria del Mar Claramont Noguès"

4 Descripció general del procés de càrrega de dades de CORDIS

Anteriorment ja s'ha comentat que l'objectiu del projecte és construir un *framework* de deduplicació que permeti crear un sistema per deduplicar les dades de CORDIS. Tot i així, tant abans com després d'aquest procés de deduplicació s'hauran de realitzar una sèrie de processos que van des d'obtenir les dades a transformar-les de cara a la seva posterior càrrega en la base de dades. La Figura 4.1 mostra quins són els passos que s'hauran de realitzar des de l'obtenció de les dades fins a la seva càrrega d'una manera genèrica.

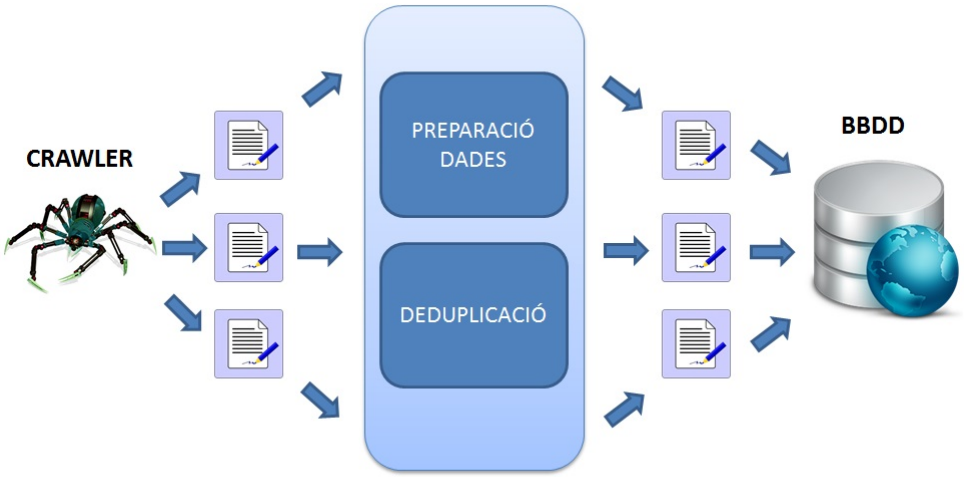


Figura 4.1: Procés general de la càrrega de dades.

Tal i com es pot veure, hi ha 4 fases generals:

- Extracció de dades (*crawler*): consisteix en l'obtenció de les dades que s'utilitzaran a partir de diferents fons d'informació, encara que la majoria de dades s'extrauran des de CORDIS.

- Preparació de les dades per la deduplicació: conjunt de preprocessos que s'hauran d'aplicar en les dades abans de poder realitzar les diferents deduplicacions.

- Deduplicació: procés en el que es decidirà quines institucions es poden considerar que són les mateixes i quines són diferents entre sí.

- Càrrega de les dades: inclou tots els passos necessaris per la càrrega d'informació a la base de dades.

Explicat breument, s'obtindran les dades mitjançant un *crawler*, que obtindrà les dades des de la pàgina web de CORDIS i les transformarà a un conjunt de fitxers. Aquests fitxers seran sotmesos a diverses transformacions de cara a possibilitar una correcte deduplicació de les dades. Tot aquest procés crearà uns nous fitxers que seràn carregats en una base de dades, a partir de la qual es podran fer les consultes necessàries pel correcte funcionament de *Sciencea*.

En aquest capítol s'explicaran amb més detall cada un d'aquests passos amb l'excepció de la descripció de les deduplicacions, que es realitzarà en el capítol 5 donada la seva particular importància dins d'aquest projecte.

4.1 Extracció de dades

Bàsicament, la informació que es requereix per cobrir les funcionalitats de *Sciencea* és tota aquella relacionada amb els projectes europeus i les institucions que realitzen aquests projectes. L'esquema dels passos que se seguiran durant el procés d'extracció de dades és el que apareix en la Figura 4.2.

La font de dades principal per al projecte és CORDIS, que, com ja s'ha comentat, és un una plataforma dedicada a informar de les diferents activitats europees de investigació i desenvolupament. És des de la seva pàgina web d'on s'extrauran gairebé la totalitat de

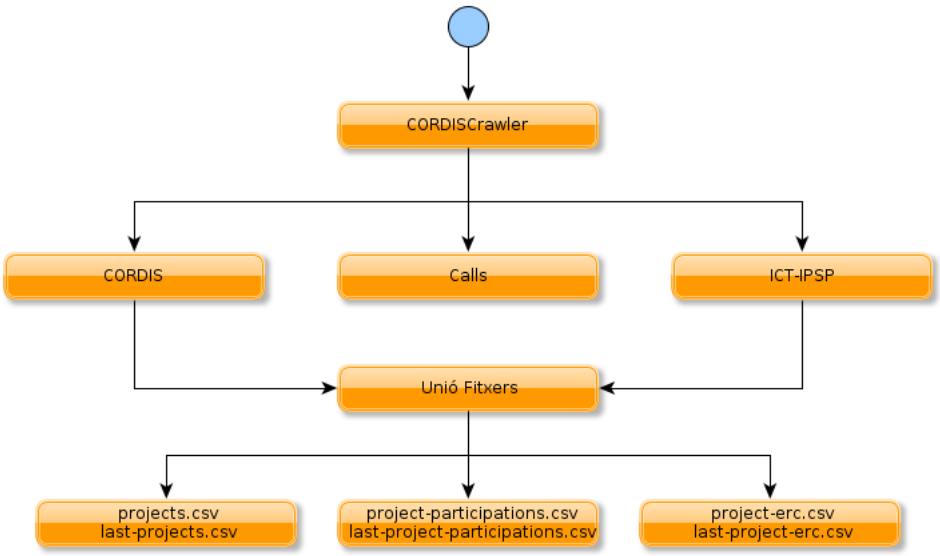


Figura 4.2: Procés d'extracció de dades

projectes i de participants que posteriorment apareixeran en la base de dades. L'obtenció d'aquestes dades es realitzarà a través d'un procés de *crawling*.

Aquest procés de *crawling*, el que farà és realitzar peticions a la pàgina web de CORDIS per obtenir tots els projectes. Com que el nombre de projectes és molt elevat, aquestes peticions es faran d'una forma paginada per a que en cada petició només es retorni la informació d'uns quants projectes. Aquesta informació es retornarà en format *xml*, de manera que s'haurà de *parsejar* per tal de passar totes les dades a fitxers *csv*.

Durant el projecte, han aparegut problemes amb la comunicació entre servidors al realitzar un gran nombre de peticions, de manera que si es detecten aquest tipus de problemes, automàticament es torna a reinicialitzar el procés de *crawling*.

La resta de projectes, que representaran menys d'un 0.5% del total, s'extrauran de la pàgina web de la comissió europea[12]. Aquests projectes estan relacionats amb tecnologies d'informació i comunicació i no apareixen en les dades de CORDIS. Per aconseguir tota aquesta informació serà necessari realitzar un *parsing* directament des del codi html de la pàgina web, intentant fer correspondre la informació obtinguda amb els camps existents en els projectes de CORDIS.

D'altra banda, una informació que no està disponible en tot aquest conjunt de dades, però que és molt interessant de cara a l'aplicació web de *Sciencea*, són les pàgines web dels diferents projectes. Per tal de trobar-les, es realitzarà una cerca a partir del buscador Google, que estarà basada en diferents criteris establerts mitjançant expressions regulars.

Així doncs, de tot aquest procés s'obtenen 2 fitxers importants, un amb la informació dels projectes (*projects.csv*) i un altre amb la informació dels participants (*project-participations.csv*). A més, es crearà un tercer fitxer anomenat *project-erc.csv*, que conté informació que es gairebé tota redundant, ja que és pràcticament igual al fitxer de projectes. Tot i així, encarà permetrà aconseguir alguna dada extra que pot ser útil, com per exemple la direcció de correu electrònic de la persona de contacte d'un projecte.

Addicionalment, també es disposa de la possibilitat de realitzar una descàrrega de dades incremental. Si es fa ús d'aquesta funcionalita,t només es buscaran els projectes des de l'última vegada que es va realitzar el *crawling*. En el cas d'utilitzar el sistema d'extracció de dades incremental, els fitxers creats rebran els noms de *last-projects.csv*, *last-project-participations.csv* i *last-project-erc.csv*. D'aquesta manera es podrà distingir amb claredat si els fitxer conté tots els projectes o només els més recents.

Finalment, també serà necessari obtenir la informació dels *calls* de projectes que realitza l'Unió Europea. Un *call* és una petició que es realitza per tal de trobar participants per a que treballin en algun projecte determinat prèviament. Tota aquesta informació també s'obtindrà des de la mateixa pàgina web de la Comissió Europea.

4.1.1 Descripció de les dades de projectes

El fitxer que conté les dades dels projectes (*projects.csv*) serà un *csv* format per uns 90000 projectes, on cada una d'aquests projectes tindrà aproximadament uns 90 atributs o camps amb informació. Tot i així, pràcticament en la totalitat de les ocasions hi haurà projectes amb atributs sense valor. Aquest fet s'haurà de tenir molt en compte posteriorment, ja que al fer la deduplicació de les dades no es podrà estar segur de la validesa d'algun dels valors obtinguts, així que no s'hi podrà confiar completament. És per això que caldrà realitzar una sèrie de processos per a preparar les dades.

D'aquests 90 atributs disponibles, només en seran utilitzats aproximadament una tercera part per a tot el procés de deduplicació i càrrega. En la Figura 4.3 es mostren alguns

dels atributs existents en les dades extretes de CORDIS. Cal tenir en compte, que dins de l'estructura de dades que utitlitzen pel projecte, també hi ha informació referent a la institució coordinadora del projecte. A continuació es disposen exemples d'aquest fet:

- Atributs referents al projecte: identificador, nom, sigles, persona de contacte, descripció, objectiu, data d'inici, data de fi, duració, cost, finançament, programa, categoria, data de modificació de dades, estat del projecte,..
- Atributs referents a la institució coordinadora: nom, país (nom i codi), regió, ciutat, adreça, pàgina web, telèfon, fax, correu electrònic..

4.1.2 Descripció de les dades de participacions

El fitxer de participacions *project-participations.csv* és el que conté tota la informació de les institucions i la relació d'aquestes amb els projectes. D'aquesta manera, el fitxer tindrà aproximadament unes 450.000 relacions de participació. Per a cada una d'elles podem trobar aproximadament uns 25 atributs.

Sempre hi haurà l'identificador del projecte amb el que es manté una relació de participació juntament amb les dades de la institució que hi participa. Entre aquestes dades hi podem trobar: nom, país (nom i codi), regió, ciutat, adreça, pàgina web, telèfon, fax, correu electrònic, latitud, longitud, tipus d'organització... En la Figura 4.3 es poden veure més atributs de les dades referents a les institucions, tot i que no hi són tots per simplicitat.

El fet que aquestes dades siguin gairebé idèntiques a les dades de les institucions coordinadores d'un projecte fa que posteriorment es puguin extreure tots els coordinadors del fitxer de projectes i unir aquestes dades amb el fitxer de participacions.

4.2 Preparació de dades

Un cop obtingudes les dades, aquestes s'hauran de tractar mitjançant diferents processos de cara a validar-les i netejar-les. En l'esquema de la Figura 4.4es pot veure què inclou aquesta fase de preparació de les dades.

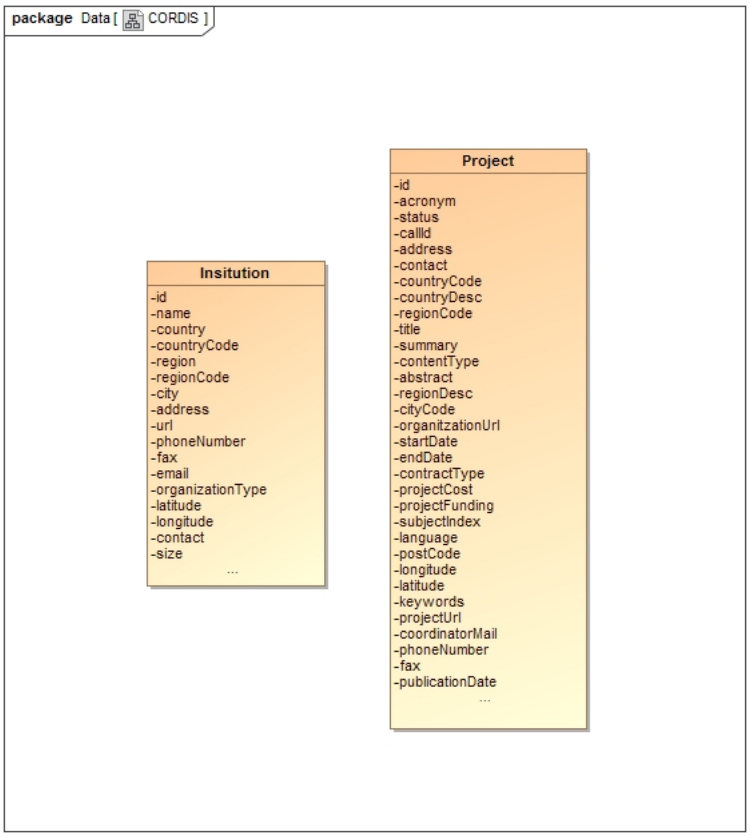


Figura 4.3: Esquema dels atributs de les dades de CORDIS

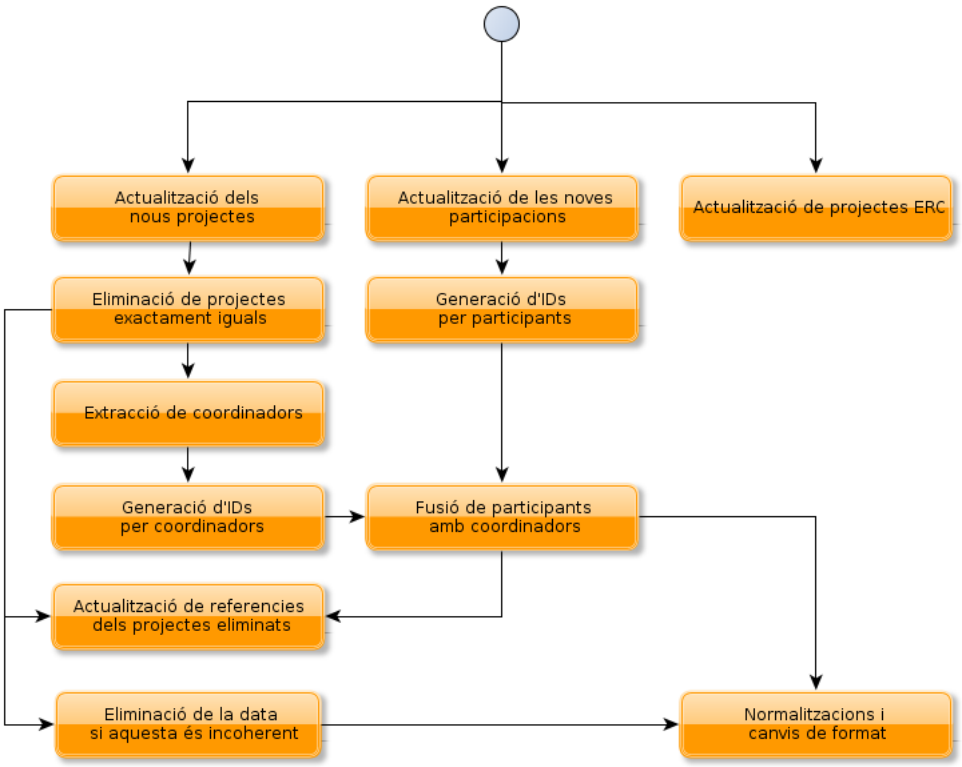


Figura 4.4: Procés de càrrega de dades.

4.2.1 Actualització dels fitxers de dades

Acabat el procés d'extracció, sempre es disposarà de dos fitxers principals, un amb projectes i l'altre amb les participacions. A més, si s'ha executat una extracció de dades incremental, també hi haurà dos nous fitxers amb els últims projectes i les participacions d'institucions en aquests últims projectes.

Així doncs, el primer que caldrà fer serà ajuntar aquests nous fitxers per tal de tenir tota la informació referent a projectes en un mateix fitxer. De manera anàloga, aquest procés es repetirà amb els fitxers d'informació referent a institucions i participacions. Cal tenir en compte que es podria donar el cas d'obtenir projectes o participacions repetides en alguna ocasió concreta, especialment si es modifica la càrrega i s'obtenen fitxers de projectes o participacions des d'una data anterior a l'última vegada que es va realitzar

Algorisme 4.1 Pseudocodi de la unió de fitxers

```
function append: void {

    File inputFile1 = projects.csv
    File inputFile2 = last-projects.csv
    File outputFile = projects-all.csv

    CSVReader reader1 = new CSVReader(inputFile1);
    CSVReader reader2 = new CSVReader(inputFile2);
    CSVWriter writer = new CSVWriter(outputFile);

    String[] line;

    while ((line = reader1.readNext()) != null){
        writer.writeLine(line);
    }

    while ((line = reader2.readNext()) != null){
        writer.writeNext(line);
    }

    reader1.close();
    reader2.close();
    writer.close();
}
```

l'obtenció de dades. L'algoritme utilitzat serà el mostrat en l'Algoritme 4.1. Bàsicament, s'aniran llegint els fitxers línia a línia i s'aniran escrivint en un fitxer de sortida sempre i quan la línia no sigui nul·la.

4.2.2 Eliminació de projectes repetits

El següent pas serà eliminar del fitxer de projectes aquells que siguin repetits. No només hi hauran repeticions pel tema de conflicte de dates d'extracció de dades, sinó també pel fet que les dades de CORDIS no són gens fiables. A més, CORDIS tracta alguns projectes que s'han renovat com si fossin projectes independents, la qual cosa no interessa. Així doncs, es consideraran projectes repetits aquells que tinguin el mateix nom exacte.

Aquests projectes repetits formaran grups de projectes duplicats que tindran un sol identificador. Així, durant aquest procés d'eliminació de repetits es guardarà una llista amb els identificadors dels projectes que s'eliminen i l'identificador comú de projecte que els hi volem assignar, per tal de poder actualitzar les referències que hi havia en la resta de fitxers on apareixien aquests projectes en concret.

Algorisme 4.2 Pseudocodi de selecció de projectes repetits

```
1  function findDuplicates: void {
2
3      File inputFile = projects-all.csv
4      File outputFile = duplicate-projects.csv
5
6      inputFile.Sort(1);       //Ordenem el fixer per la columna 1, referent al nom
           del projecte
7
8      CSVReader reader = new CSVReader(inputFile);
9      CSVWriter writer = new CSVWriter(outputFile);
10
11     String[] line;
12     String lastName = "";    //Nom del projecte, corresponent a la columna 1 del
           fitxer projects-all
13     String lastRef = "";     //Identificador del projecte, corresponent a la
           columna 0 del fitxer projects-all
14
15
16     while ((line = reader1.readNext()) != null) {
17         if (line[1].equals(lastName)){
18             String[] writeLine = new String[2];
19             writeLine[0] = lastRef;
20             writeLine[1] = line[0];
21             writer.writeNext(writeLine);
22         }
23         else {
24             lastName = line[1];
25             lastRef = line[0];
26         }
27     }
28
29     reader.close();
30     writer.close();
31 }
```

L'algorisme utilitzat serà el representat en l'Algorisme 4.2. El que es realitzarà serà ordenar el fitxer de projectes segons el nom del projecte i recórrer aquest fitxer ordenat. Sempre que es trobi un projecte que té el mateix nom que el de la línia anterior, s'escriurà en el nou fitxer de duplicats l'identificador dels dos projectes que tenen el mateix nom, posant en primer lloc el que ha aparegut primer, i que serà l'identificador del projecte que es considerarà com a bo.

Una vegada obtingut aquest fitxer amb les parelles d'identificadors de projectes duplicats, es recorrerà el fitxer de projectes i s'eliminaran del fitxer aquells projectes que estiguin en la columna 1 del fitxer de duplicats. Per saber si un projecte està en aquesta columna del

Algorisme 4.3 Pseudocodi d'eliminació de projectes repetits

```
function findDuplicates: void {

    File duplicateFile = duplicate-projects.csv
    File projectsFile = projects-all.csv

    File outputFile = projects-no-duplicates.csv

    CSVReader readerDuplicates = new CSVReader(duplicateFile);
    CSVReader readerProjects = new CSVReader(projectsFile);
    CSVWriter writer = new CSVWriter(outputFile);

    String[] line;
    HashSet<String> set = new HashSet<String>();

    while ((line = readerDuplicates.readNext()) != null) {
        set.add(line[1]);
    }

    while ((line = readerProjects.readNext()) != null) {
        if (!set.contains(line[0]){
            writer.writeNext(line);
        }
    }

    readerDuplicates.close();
    readerProjects.close();
    writer.close();
}
```

fitxer de duplicats, prèviament es carregaran en un *set*, tots els identificadors duplicats, tal i com es mostra en l'Algorisme 4.3.

4.2.3 Extracció de coordinadors

La informació referent a les institucions que coordinen els diferents projectes està inclosa en el fitxer de projectes. Així doncs, serà necessari extreure aquesta informació (nom de la institució, país, regió, ciutat, web, mail de contacte...) per poder-hi treballar. D'aquesta manera, es crearà un nou fitxer de coordinadors que seguirà el mateix format que el de participacions, ja que en els dos casos es tracta d'institucions, i per tant, pot ser que sigui necessari aplicar un mateix procés als dos fitxers. Si no s'obtenen els coordinadors d'aquesta manera, es perdrien aquestes institucions com a participants del projecte, ja que no apareixen amb la resta d'institucions que treballen en un projecte.

L'algorisme que s'utilitzarà serà simplement recórrer el fitxer, obtenint els valors que es necessiten i posant-los d'una manera ordenada en un fitxer de coordinadors seguint l'esquema del fitxer de participacions.

4.2.4 Generació d'identificadors

Un dels problemes més importants en les dades que s'obtenen de CORDIS és el fet que no es pot confiar en els valors del camp d'identificadors que proporcionen. Un mateix identificador pot estar associat a institucions diferents, i a més, una mateixa institució també pot tenir diversos identificadors. Aquest és el problema principal a resoldre pel qual s'ha creat aquest projecte de final de carrera, ja que de ser correcta aquesta informació, tota la deduplicació d'institucions no seria necessària.

És per això que el següent pas serà donar un identificador a cada una de les institucions, ja siguin del fitxer de participants o de coordinadors. Conseqüentment, s'establirà que hi ha una institució per a cada participació o per cada relació de coordinació. És per això que posteriorment s'haurà de realitzar una deduplicació d'institucions, ja que una mateixa institució pot ser participant o coordinador de diferents projectes.

L'algorisme utilitzat serà molt senzill, tal i com es pot veure en l'Algorisme 4.4. Es recorrerà el fitxer corresponent i es canviarà el valor del camp identificador per un nombre que s'anirà incrementant des d'un nombre inicial donat. El nombre inicial serà diferent per participants i per coordinadors per assegurar que no es repeteixin.

4.2.5 Fusió de participants amb coordinadors

Es podria donar el cas que una institució que és coordinadora d'un projecte no tingués cap relació de participació amb cap altre projecte. Així doncs, per tal de garantir que hi haurà totes les institucions en un mateix fitxer s'hauran de fusionar els fitxers de coordinadors i de participacions. Aquest procés es farà de manera anàloga a l'explicat en l'apartat 4.2.1, creant un fitxer amb el nom de *institutions-plus-coordinations.csv*.

4.2.6 Actualització de referències

Eliminar alguns projectes per estar repetits causarà que el nou fitxer amb totes les institucions sigui incorrecte, ja que hi hauran institucions que seran participants d'un projecte

Algorisme 4.4 Pseudocodi de creació d'identificadors.

```
1  function createId: void {
2
3      File inputFile = coordinators.csv
4      File outputFile = coordinators-id.csv
5
6      CSVReader reader = new CSVReader(inputFile);
7      CSVWriter writer = new CSVWriter(outputFile);
8
9      String[] line;
10     int idNumber = 0;
11
12     while ((line = reader1.readNext()) != null) {
13         line[3] = idNumber;      //L'identificador és representat en la columna 3
               del fitxer
14         writer.writeNext(line);
15         idNumber++;
16     }
17
18     reader.close();
19     writer.close();
20 }
```

que s'ha eliminat. Per solucionar això, en el fitxer de participacions es modificarà el valor de l'identificador de projecte d'aquells projectes eliminats, i es canviarà per l'identificador del projecte del qual era duplicat. Això es realitzarà a partir de la llista creada a l'eliminar projectes. Si existeix alguna participació repetida, és a dir, que hi hagi més d'una parella igual de l'estil "identificador de projecte - identificador d'institució", s'eliminarà del fitxer per tal de garantir que s'obtenen parelles úniques.

Així doncs, l'algorisme utilitzat serà el de l'Algorisme 4.5. En primer lloc es carregaran en un Map les parelles d'identificadors amb els projectes duplicats. Aleshores, es recorrerà el fitxer de participacions i si es troba una referencia a un projecte eliminat (i que per tant està en el map), es modificarà aquesta referència, obtenint un nou fitxer amb totes aquestes referències actualitzades.

4.2.7 Eliminació de la data si és incoherent amb el programa

En les dades es troben projectes en que la seva data d'inici no es correspon amb el període del programa europeu del que forma part. En aquests casos ,ja sigui perquè la data d'inici

Algorisme 4.5 Pseudocodi de l'actualització de referències.

```
function findDuplicates: void {

    File duplicateFile = duplicate-projects.csv
    File institutionsFile = institutions-plus-coordinations.csv

    File outputFile = institutions-plus-coordinations-projects.csv

    CSVReader readerDuplicates = new CSVReader(duplicateFile);
    CSVReader readerinstitutions = new CSVReader(institutionsFile);
    CSVWriter writer = new CSVWriter(outputFile);

    String[] line;
    HashMap<String,String>() map = new HashMap<String,String>();

    while ((line = readerDuplicates.readNext()) != null) {
        map.put(line[1], line[0]);
    }

    while ((line = readerInstitutions.readNext()) != null) {
        if (map.containsKey(line[0]){
            line[0] = map.get(line[0]);
        }
        writer.writeNext(line);
    }

    readerDuplicates.close();
    readerinstitutions.close();
    writer.close();
}
```

Programa	Any Inici	Any fi
FP1	1984	1988
FP2	1987	1991
FP3	1990	1994
FP4	1994	1998
FP5	1998	2002
FP6	2002	2006
FP7	2007	2013
FP8	2014	2020

Taula 4.1: Relacions entre programa i dates acceptades.

del projecte és anterior a la data de l'inici del programa o posterior a la data de final del programa, s'eliminaran les dates incorrectes. Així s'evitaran crear inconsistències en la base de dades que es crearà a posteriori. La taula 4.1 mostra el període d'anys acceptat per a cada un dels diferents tipus de programa.

L'algorisme utilitzat consistirà en recórrer el fitxer de projectes i comprovar per a cada projecte si la data del projecte està inclosa en el període d'anys definit segons el seu programa, eliminant aquells projectes que no ho compleixin.

4.2.8 Normalitzacions i canvis de format

De cara a obtenir unes dades que sempre siguin iguals, es realitzarà un procés de normalització en diferents camps. A més, com trobem formats diferents en les dates, aquestes es transformaran per obtenir un format únic que es pugui carregar en la base de dades. També es realitzaran alguns altres canvis menors en les dades per a que puguin ser carregades.

Algunes d'aquestes transformacions són el canvi a minúscules, l'eliminació d'espais duplicats o l'eliminació d'accents, i s'aplicaran només a uns pocs camps determinats que s'utilitzaran posteriorment. En alguns casos, es mantindrà el camp original i s'afegira un camp extra amb la normalització aplicada.

Un exemple de normalització seria:

- "Universitat Politècnica de Catalunya - (U.P.C)" –> "universitat politecnica de catalunya upc".

4.3 Càrrega de dades

Un cop ja es disposi de tota la informació deduplicada, caldrà carregar-la a la base de dades. En aquest cas, la base de dades on es carregaran tota la informació serà *Dex*[1], la base de dades que es desenvolupa en el grup DAMA-UPC. Aquesta base de dades està especialment pensada per treballar amb grans volums de dades, de manera que és molt adequada per aquest projecte. Una de les característiques principals de *Dex* és el fet de ser una *Graph Database*. Això implica que està estructurada en forma de nodes i arestes, que representaran diferents elements i les relacions existents entre aquests.D'aquesta manera, les institucions i els projectes es guardaran com a nodes dins de *Dex*, mentre que les relacions de participació o de coordinació entre un projecte i una institució seran representades mitjançant arestes.

Així doncs, aquest procés de càrrega, sovintment conegut com procés *ETL* (*Extract Transform Load*) fa servir un component del grup DAMA-UPC que permet carregar les dades de dues formes diferents:

- En primer lloc, permet carregar les dades des d'una classe *Java* directament, de manera que es podran fer insercions en la base de dades mentre es realitzen diferents càlculs.

- En segon lloc, també permet carregar les dades directament des d'un arxiu directament mitjançant un script. Aquesta és la manera més eficient i la que s'utilitzarà en aquest projecte sempre que sigui possible.

Un exemple del *script* que s'utilitza per a realitzar la càrrega de les dades seria el mostrat a continuació. En aquest es pot veure la càrrega com a nodes de les institucions. El *script* agafarà el fitxer "*project-participations-uniq.csv*", el separarà en columnes segons el caràcter "|" i inserirà a la base de dades el valor de cada columna a l'atribut corresponent del tipus de node *Institution*. De la mateixa manera es realitzarà la càrrega de les arestes de la relació *coordinates*: se separarà el fitxer "coordinations-replaced" en columnes i s'indicarà quina columna és l'identificador del node projecte i quin es el de la institució amb el que el volem relacionar.

```
LOAD NODES './src/test/resources/data/project-participations-uniq.csv'
COLUMNS *,name,*,code,*,*,contactInformation,url,address,*,country,region,city
    ,*,*,*,*,*,*,*,*,*,*,*,*,*
INTO Institution
FIELDS TERMINATED '|' mode rows

LOAD EDGES './src/test/resources/data/coordinations-replaced.csv'
COLUMNS target,*,*,src,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*
INTO coordinates ignore src, target
WHERE TAIL src = Institution.code HEAD target = Project.id
FIELDS TERMINATED '|' mode rows
```

D'altra banda, l'esquema resultant en la base de dades de Dex serà el mostrat en la Figura 4.5 . En aquest cas, no es mostra tota la informació que hi haurà en la base de dades, tant pel que fa a les classes com als seus atributs, sinó que s'abreujarà per tal de nomes incloure aquella informació relacionada directament amb la deduplicació de les dades de CORDIS.

Com es pot observar en el diagrama de classes, la classe *Institution* tindrà dos relacions cap a la classe *Project*. La primera d'elles serà la relació *participate*, que indica que una institució està present en un projecte. La segona relació serà la de *coordinates,* que indica que una institució és la coordinadora del projecte. En el primer cas, podran haver-hi n institucions per a cada projecte com a participants, mentre que en el cas dels coordinadors, només hi podrà haver una institució per projecte. A més, una institució podrà tenir un o més *Acronym*, que seràn possibles formes d'abreujar el nom d'aquella institució. D'altra banda, també es guardaran com a classes independents els països, les regions, i els programes europeus d'investigació als que pertany un projecte.

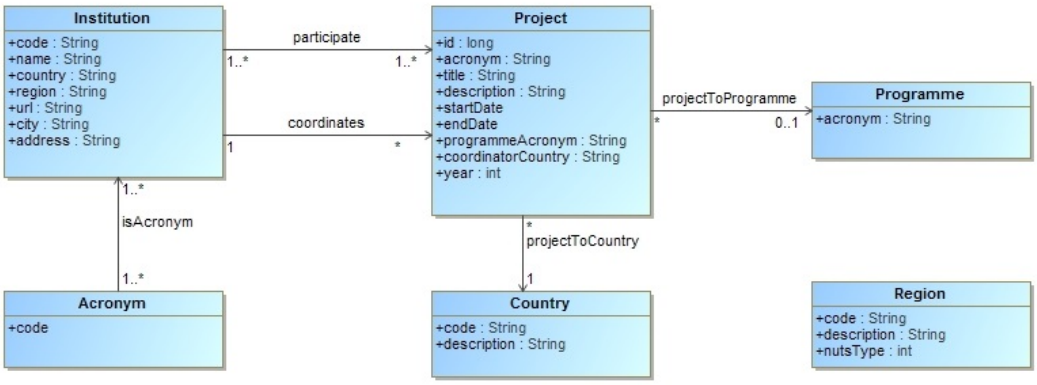


Figura 4.5: Esquema de la base de dades de *Dex*.

5 Procés de deduplicació de dades de CORDIS

5.1 Descripció general

Aquesta és segurament la part més complexa de tot el procés fins a obtenir tota la informació disponible en una base de dades. L'objectiu de la mateixa és obtenir una llista d'institucions úniques, sense que n'hi hagi de repetides. S'ha definit que perquè dues institucions es considerin iguals, aquestes han de tenir una semblança molt elevada tan en el nom com en els seus atributs i, a més, han de pertànyer a una mateixa ciutat. D'aquesta manera, dues institucions amb el mateix nom, país i regió seran considerades diferents si son de diferents ciutats.

En la Figura 5.1 es pot veure un exemple de institucions que es volen obtenir juntes o per separat. En el cas de 1 i 2, es voldrà considerar que es tracta de la mateixa institució, ja que, encara que tenen adreça diferent, són de la mateixa ciutat i altres atributs s'assemblen. En canvi, la institució 3 haurà de ser considerada com una institució diferent de 1 i 2 al ser d'una altra ciutat. Més endavant, ja es veurà amb detall quin són els criteris exactes per considerar dues institucions iguals o diferents.

Així doncs, per aconseguir una bona deduplicació de les institucions, és clar que es necessitarà garantir una certa correcció en els camps de regió i de ciutat. Com això no succeeix dins de les dades de CORDIS (són molt freqüents casos de regions i ciutats escrites de varies maneres diferents, com podria ser el cas de "Catalunya", "Cataluña", "Catalunia", ...) es fa necessari realitzar abans de tot una deduplicació tant de regions com de ciutats per tal de poder realitzar una comparació més ajustada amb les institucions.

Una vegada es tinguin els fitxers preparats correctament ja es podrà procedir al procés de deduplicació d'institucions, que es farà de dues formes diferents:

Num	Nom	País	Ciutat	Direcció
1	Universitat Politècnica de Catalunya	ES	Barcelona	Carrer de Jordi Girona, 31, 08034 Barcelona
2	Universitat Politècnica de Catalunya	ES	Barcelona	Avinguda Diagonal, 649, 08028 Barcelona
3	Universitat Politècnica de Catalunya	ES	Terrassa	Colon, 11, 08222, Terrassa

Figura 5.1: Exemple de dades extretes directament de CORDIS.

- Deduplicació per nom: a partir del nom de les institucions, es buscaran les institucions duplicades utilitzant un sistema de *sliding window*, on les institucions estaran ordenades per ordre alfabètic.

- Deduplicació per sigles o acrònims: es buscaran les possibles sigles de totes les institucions. Aleshores, es faran grups d'institucions que podrien tenir una mateixa sigla. Per a cada un d'aquests grups, es realitzaran comparacions de tots amb tots. Així doncs, s'utilitzarà un mètode de *standard blocking* .

A partir d'aquestes dues deduplicacions, es crearan grups d'institucions que seran considerades com a una única institució. De cada grup, s'escollirà una institució representant, que portarà el nom més freqüent dins del grup i que serà aquella que tingui més atributs vàlids. D'aquesta manera, la informació d'aquesta institució representant serà la que es carregui a la base de dades.

A continuació, es canviaran totes les referències a les institucions que s'eliminen per considerar-les repetides, i en el seu lloc es posaran els identificadors de les institucions que són representants en el seu grup. Aquest procés d'actualització de referències s'haurà de fer tant en el fitxer de participacions com en el de coordinadors o en el fitxer que es relaciona una institució amb les seves possibles sigles. De fet, aquest procés és molt similar al realitzat a l'actualitzar les referències de les institucions pels projectes eliminats anteriorment per considerar-los repetits.

Tot aquest procés es pot veure representat d'una manera general en la Figura 5.2.

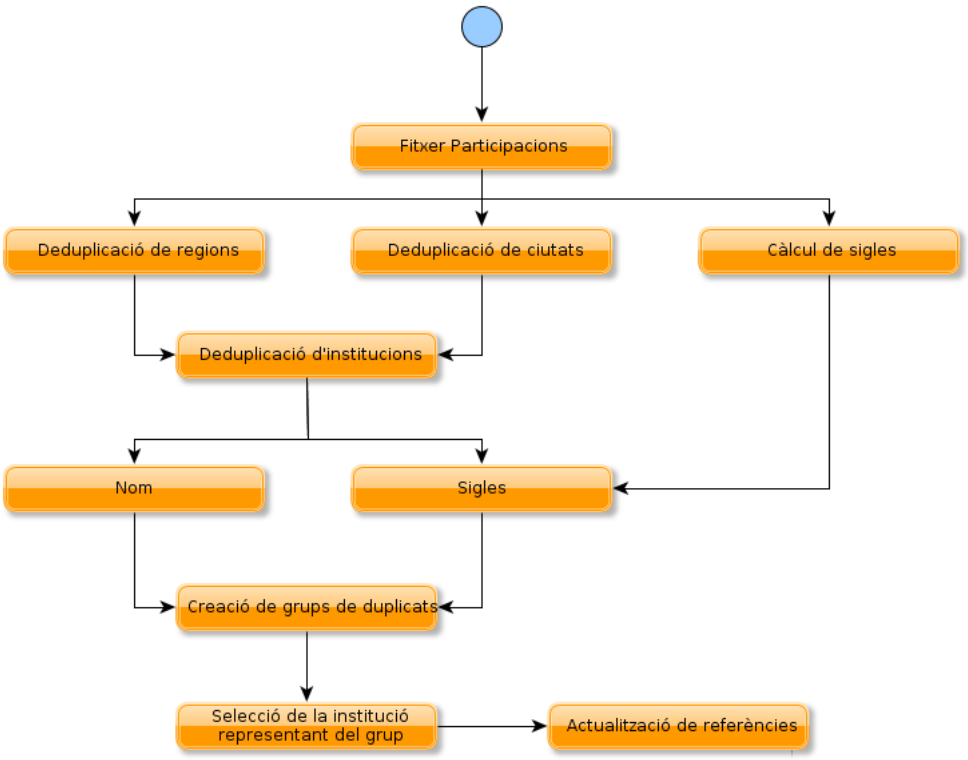


Figura 5.2: Procés general de la deduplicació.

5.2 Deduplicació de regions

La deduplicació de regions és la més senzilla que es realitzarà en aquesta part del procés. Per realitzar-la, s'utilitzara *Dexdup*, el *framework* dissenyat per aquest tipus de tasques. L'esquema que se seguirà és el representat en la Figura 5.3.

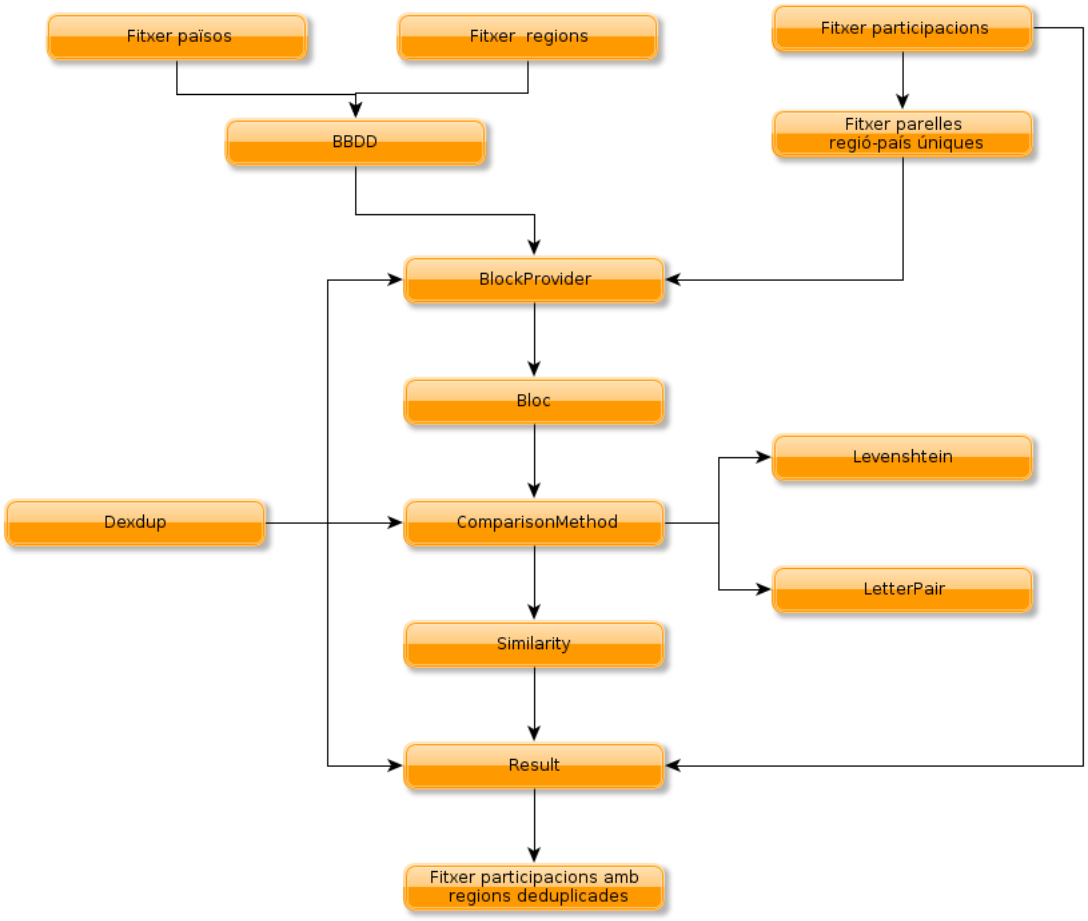


Figura 5.3: Procés de la deduplicació de regions.

En primer lloc, es carregaran en la base de dades com a nodes el conjunt de tots els països, totes les regions i els seus codis. Aleshores, a partir del fitxer de participacions, s'agafaran totes les parelles úniques regió-país que apareixen en les dades i es posaran en un nou fitxer. Aquest fitxer serà el que entrarà al *blockIterator* que s'encarregarà de realitzar els blocs per fer les comparacions.

Cada bloc tindrà com a primer element a la primera parella de l'estil regió-país llegida des del fitxer d'entrada. La resta d'elements del fitxer seran el conjunt de parelles inserides a la base de dades de l'estil regió-país que tinguin el mateix país que el primer element del bloc. Així, es compararà el valor del camp regió del fitxer de participacions amb totes

Algorisme 5.1 Pseudocodi del *BlockIterator*

```
String[] line;
ArrayList<Object> block = new ArrayList<Object>()

function hasNext(): boolean {
    File file = participacions.csv
    CSVReader reader = new CSVReader(file);
    line = reader.readNext();
    return (line!=null);
}

function next(): ArrayList<Object> {
    String country = line[0];
    String region = line[1];
    block.add(region):
    Objects o = BBDD.selectAllRegionsInCountry(country);
    for (Object obj: o) {
        block.add(obj.getRegion());
    }
    return block;
}
```

les regions possibles del país que coincideix amb el camp país del fitxer. Aquestes regions s'obtindran fent una crida a la base de dades.

D'aquesta manera tindrem w blocs de $n+1$ parelles regió-país, on n serà el nombre de regions d'un país donat i w serà el nombre de parelles úniques regió-país que hi haurà en el fitxer de participacions. En l'Algorisme 5.1 es pot veure el pseudocodi que executaria el *BlockIterator*.

Una vegada ja s'ha creat un bloc, s'iniciaran les comparacions entre les parelles d'aquell conjunt de registres. Per cada bloc es comparará la primera entitat contra totes les altres.

Per fer aquesta comparació, s'utilitzaran 2 comparadors de *Strings* diferents: *Levenshtein*[13] i *LetterPair*. Així doncs, es realitzarà la comparació per cada un d'aquests dos comparadors i se'n retornarà el valor màxim.

5.2.1 LetterPair

LetterPair és un comparador de *Strings* que calcula la similitud basant-se en les vegades que es repeteixen combinacions de dues lletres. D'aquesta manera, de cada una de

Algorisme 5.2 Pseudocodi del compadaror *LetterPair*

```
String[] line;

function compareLetterPair(String str1, String str2): double {

    ArrayList<String> pairs1 = wordLetterPair(str1);
    ArrayList<String> pairs2 = wordLetterPair(str2);

    int union = pairs1.size89 + pairs2.size();
    int intersection = 0;

    for (int i = 0; i<pairs1.size(); i++) {
        Object pair1 = pairs1.get(i);
        for (int j = 0; j<pairs2.size(); j++) {
            Object pair2 = pairs2.get(j);
            if(pair1.equals(pair2)){
                intersection++;
                pairs2.remove(j);
                break;
            }
        }
    }

    return (2*intersection)/union;
}
```

les paraules d'un *String* se n'extreuen totes les parelles de lletres consecutives per tal de comparar-les posteriorment amb les parelles de l'altre *String*. El resultat de la comparació vindrà donat pel nombre de parelles de lletres d'una cadena que s'ha trobat en l'altre, tenint en compte que un cop trobada una parella de lletres, aquesta s'eliminarà de la llista. Aquest nombre de parelles serà multiplicat per dos i dividit entre la suma del número de parelles de lletres de cada un dels *Strings*. El pseudocodi d'aquest comparador es pot veure en l'Algorisme 5.2.

5.2.2 Levenshtein

D'altra banda, el comparador *Levenshtein* es basa en la *Distància de Levenshtein*. Aquest valor indica el nombre de canvis elementals necessaris per passar d'una cadena de caràcters a una altre. Per canvis elementals s'entenen la inserció i l'eliminació d'un caràcter, a més del canvi d'un caràcter per un altre. Per a trobar el valor de la similitud final, el comparador *Levenshtein* dividirà el nombre de canvis elementals requerits per passar d'un *String* a un altre entre el nombre de caràcters de la cadena de caràcters més llarga

de les que es comparen.

L'algorimse utilitzat és una optimització respecte l'algorisme tradicional. En aquest algorisme tradicional es crea una matriu de *size1* + 1 columnes i *size2* + 1 files (on *size1* és la longitud del primer string, i *size2* és la longitud del segon), i s'inicialitza la fila superior i la columna de l'esquerra amb valors consecutius de de 0 fins a *size1*, i de 0 fins a *size2* respectivament. Un cop inicialitzada, es recorrerà fila a fila la matriu i en cada posició *(n,m)* es posarà el valor mínim entre:

- (n-1,m) + 1: situació en què s'elimina una lletra
- (n, m-1) + 1: situació en què s'afegeix una lletra
- (n-1, m-1) + *cost*: situació en què se substitueix una lletra. La variable de *cost* serà 1 si les lletres situades en les posicions n i m de les paraules són diferents i 0 si són iguals.

A continuació es veurà el procés de comparar les paraules "BONA" i "HOLA" amb algun exemple concret de com s'omple la matriu, que es pot veure completada en la Figura 5.4.

		H	O	L	A
	0	1	2	3	4
B	1	1	2	3	4
O	2	2	1	2	3
N	3	3	2	2	3
A	4	4	3	3	2

Figura 5.4: Matriu creada amb l'algoritme tradicional de *Levenshtein*

Per exemple, per a omplir la posició (2,1) de d'aquesta comparació (corresponent a les lletres "H" de "HOLA" i "O" de "BONA"), es tindran en compte els tres casos anteriors:

- (1,1) + 1 = 1 + 1 = 2
- (2, 0) + 1: 2 + 1 = 3
- (1,0) + *cost* : 1 + 1 (perquè són diferents) = 2

D'aquesta manera, a la posició (2,1) es posarà el valor 2, que és el mínim dels tres valors.

En el cas de voler omplir la posició de la matriu (2,2), que es correspon amb les lletres "O" de les dues paraules, es realitzaran els següents càlculs:

- (1,2) + 1 = 2 + 1 = 3
- (2,1) + 1 = 2 + 1 = 3
- (1,1) + *cost* : 1 + 0 (perquè els caràcters són iguals) = 1

Així doncs, el valor del cost en la posició (2,2) seria 1.

Un cop realitzat aquest procés per a totes les posicions de la matriu, el valor de la distància de *Levenshtein* serà el de la posició (4,4), que es correspon amb le valor 2. Aixó significa que a partir de la paraula "HOLA", realitzant 2 insercions, eliminacions o substitucions de caràcters es pot arribar a la paraula "BONA". Podem comprovar que el resultat és correcte ja que si substituim "H" i "L" per "B" i "N", passem d'una paraula a l'altre, i no hi ha cap manera de fer-ho amb menys operacions.

L'algorisme de *Levenshtein* implementat en el projecte es mostra en l'Algorisme 5.3. Aquest algorisme segueix la mateixa idea que l'anterior però evita emmagatzemar tota la matriu sencera. Només es guardarà la fila anterior a la fila sobre la que s'estan calculant els valors i els valors de la fila actual. Al final de tot s'obtindrà la distància de *Levenshtein* de forma anàloga a l'exemple anterior. El resultat de la comparació de strings serà aquest valor dividit entre la longitud màxima dels dos strings d'entrada. En l'exemple utilitzat entre "HOLA" i "BONA", la similitud seria:

$$\frac{levenshteinDistance}{Max(\text{“}HOLA\text{”}.length,\text{“}BONA\text{”}.length)} = \frac{2}{Max(4,4)} = \frac{2}{4} = 0.5$$

Algorisme 5.3 Pseudocodi del compadaror *Levenshtein*

```
String[] line;

function compareLevenshtein(String str1, String str2): double {

    int size1 = str1.length();
    int size2 = str2.length();

    if ((size1 == 0) || (size2 == 0)) {
        return 0;
    }

    int[] previusCost = new int[size1+1];
    int[] actualCost = new int[size2+1];

    int[] aux;

    int i;  //iterador sobre str1
    int j;   //iterador sobre str2
    char c; //caracter j de str2
    int cost = 0;

    for (i = 0; i <= size1; i++) {
        previousCost[i] = i;
    }

    for (j = 1; j <= size2; j++) {
        c = str2.charAt(j - 1);
        actualCost[0] = j;

        for (i = 1; i <= size1; i++) {
            if (str1.charAt(i-1) == c){
                cost = 0;
            }
            else{
                cost = 1;
            }

            actualCost[i] = Math.min(Math.min(actualCost[i - 1] + 1, previousCost[
                i] + 1), previousCost[i - 1] + cost));
        }
        aux = previousCost;
        previousCost = actualCost;
        actualCost = aux;
    }

    return (1.0 - (previousCost[size1] / Math.max(str1.length(), str2.length())));
}
```

5.2.2.1 Tractament dels resultats de les comparacions

Per tractar els resultats obtinguts es crearà un map que tindrà com a conjunt d'entrades, el conjunt de parelles regió-país úniques del fitxer d'entrada de participacions. Cada una d'aquestes tindrà com a valor el registre regió-país amb el que s'hagi obtingut una similitud més elevada juntament amb aquesta similitud.

Així doncs, cada vegada que es realitzi una comparació, es mirarà si la nova similitud és major que la existent per aquella entrada, i de ser així se substituirà el seu valor per la parella regió-país amb la que s'ha comparat juntament amb la nova similitud.

Una vegada s'hagin realitzat totes les comparacions, s'actualitzarà el fitxer de participacions. Aquest fitxer s'anirà llegint línia a línia, i per cada una d'elles es buscarà la seva parella regió-país en el map de resultats. Si la similitud supera un llindar que s'ha situat en 0.6 canviarem el valor de la regió en el fitxer pel de la regió amb la que s'ha comparat. D'aquesta manera acabarem obtenint un nou fitxer de participacions amb les regions ja deduplicades.

El fitxer de configuració de Dexdup utitlitzat serà el següent:

```
<config>

  <deduplication name="regionInstitutions-normalizer" active="true">

    <blockProvider bean="RegionsByCountryBlockIterator" blockSize="100">
      <property name="inputFile" value="institutions-region-country.csv"/>
      <property name="separator" value="|" />
      <property name="quoteChar" value="\0" />
    </blockProvider>

    <comparisonMethodname="MultipleComparatorsInst" bean="
        BasicOperatorCompositeComparable">
      <property name="operator" value="max"/>
      <propertyComparator name="Levenshtein" comparator="RegCountryLevenshtein">
        <property name="removeAccents" value="true"/>
      </propertyComparator>
      <propertyComparator name="LetterPair" comparator="RegCountryLetterPair">
        <property name="removeAccentsEnables" value="true"/>
      </propertyComparator>
    </comparisonMethod>

    <result bean="RegionsNormalizerResult">
      <property name="threshold" value="0.6"/>
      <property name="inFileInst" value="project-participations-city.csv"/>
```

```
      <property name="outFileInst" value="project-participations-regions.csv"/>
      <property name="separator" value="|" />
      <property name="quoteChar" value="\0" />
    </result>

  </deduplication>

</config>
```

5.3 Deduplicació de ciutats

5.3.1 Descripció general

Com ja s'ha dit, ens trobem que, igual que passa amb les regions, les dades referents a les ciutats que hi ha en els fitxers procedents de CORDIS no són gens fiables. A més, el conjunt de possibles ciutats és molt més gran que el de possibles regions, de manera que complica de gran manera la deduplicació. L'esquema d'aquesta deduplicació es pot trobar en la Figura 5.5. A grans trets, la deduplicació de ciutats funcionarà de la següent manera: es carregaran dades referents a les ciutats i als noms que poden rebre (alternatives) procedents d'un fitxer obtingut a traves de Geonames [14]. Aleshores, es buscaran les correspondències entre les ciutats que apareixen en el fitxer de participacions i les ciutats carregades en la base de dades. Un cop trobades, el que es farà es substituir les ciutats de les dades de CORDIS per la ciutat més similar de les de Geonames. A més, les ciutats de les quals no es trobi cap correspondència seran eliminades.

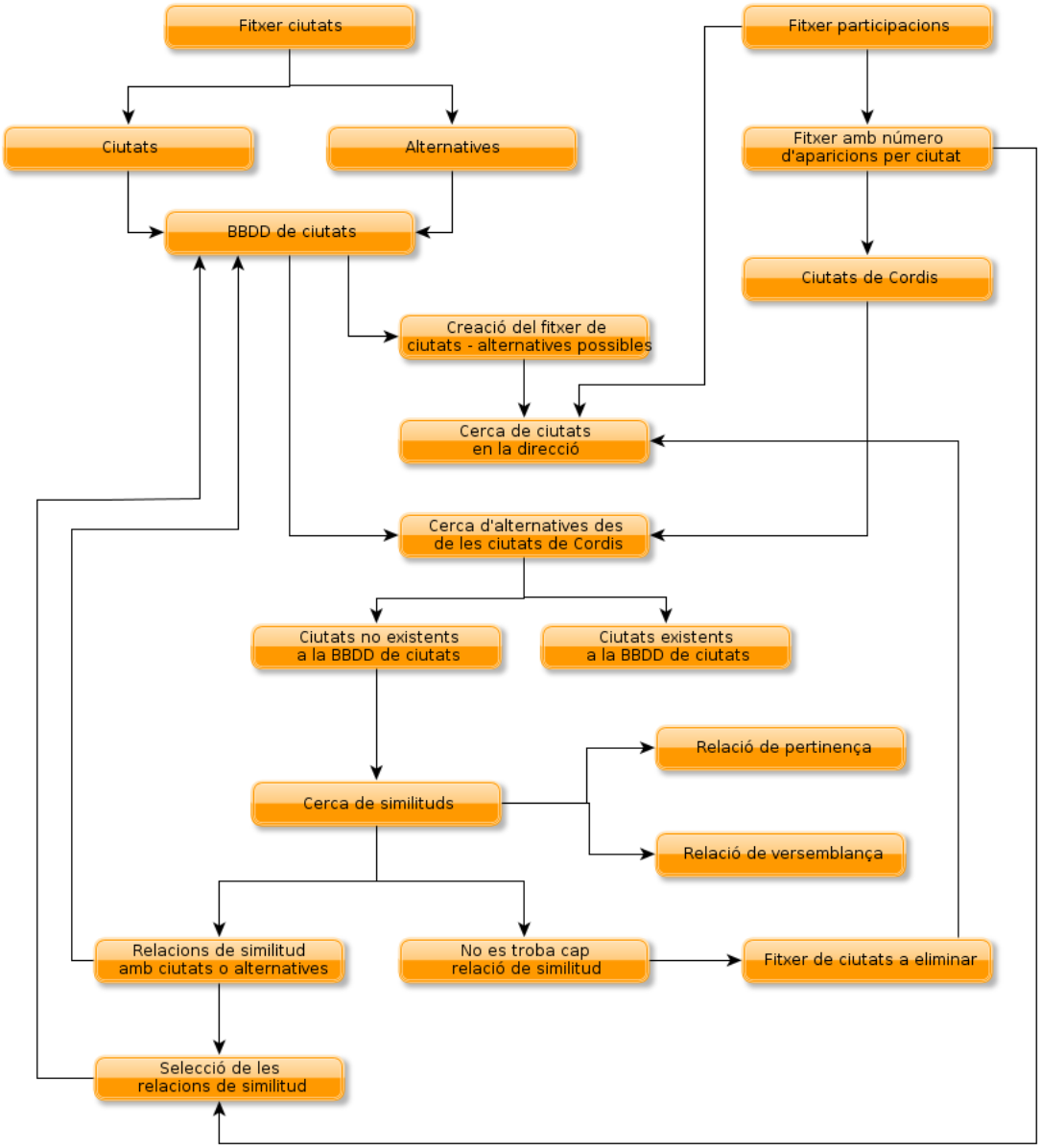
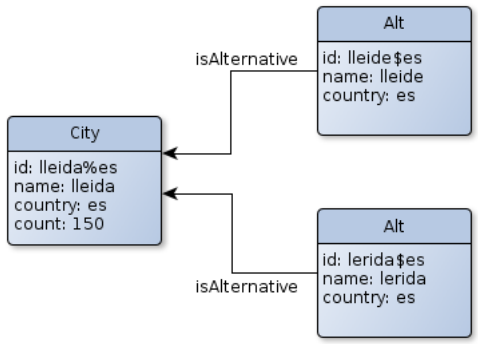


Figura 5.5: Procés de la deduplicació de ciutats.

5.3.2 Creació i càrrega de la base de dades de ciutats

Dins de la deduplicació de ciutats, el primer que s'haurà de fer es crear i carregar dades a una base de dades específica per a les ciutats, que al utilitzar *Dex*, estarà organitzada en forma de graf. En aquesta base de dades hi podrà haver nodes de tipus *City* i nodes de tipus *Alt*. A més, hi haurà un sol tipus d'arestes anomenat *isAlternative*, que relacionarà una alternativa *Alt* amb una ciutat *City*. Una alternativa serà una paraula que s'utilitza per referir-se a una ciutat. Aquests dos tipus de nodes tindran un nom, un país i un identificador composat per la concatenació del nom i el país. A més, el tipus *City* també tindrà un atribut que contindrà el nombre de vegades que apareix aquella ciutat o una de les seves alternatives en les dades de CORDIS.

Un exemple de l'organització del graf seria la de la Figura 5.6, on es pot veure que hi ha "Lleida" com a *City*, "Lérida" i "Lleide" com Alt, i les corresponents relacions de tipus *isAlternative*.



enshtein

Figura 5.6: Exemple de les relacions en la base de dades de ciutats

Per a realitzar la deduplicació de ciutats es partirà de 2 fitxers diferents. Un serà el fitxer de participacions provinent de les dades de CORDIS i l'altre serà un fitxer extern, provinent de Geonames. Aquest segon fitxer contindrà el conjunt de totes les poblacions a nivell mundial, el seu país i un conjunt d'alternatives a aquestes poblacions.

Així doncs, el primer pas serà carregar en la base de dades, les ciutats que apareguin en el fitxer de Geonames, les alternatives existents i les relacions entre una ciutat i totes les

seves possibles alternatives. Al realitzar aquest pas, ja es validarà que no es carreguin ciutats repetides (es consideraran repetides si tenen el mateix nom i són del mateix país) i que tota la informació estigui normalitzada.

El format de les dades del fitxer de Geonames seria el mostrat en la taula de la Figura 5.7:

Ciutat	**Alternatives**	**País**
Barcelona	Bar, BCN, Barcellona, Barcino, Barselona, la Ciudad Condal	ES

Figura 5.7: Exemple de les dades de Geonames

5.3.3 Càlcul del nombre d'aparicions d'una ciutat

Una de les dades que es necessitarà per a realitzar la deduplicació per ciutats, és el nombre d'aparicions d'una ciutat en el fitxer de participacions.

Així doncs, en primer lloc serà necessari obtenir quantes vegades apareix cada valor del camp ciutat de les dades de CORDIS. Això es buscarà a partir del fitxer de participacions. Tal i com es mostra en l'Algorisme 5.5, s'anirà llegint el fitxer línia a línia, inserint en un map les possibles ciutats (amb el seu país concatenat) com a clau i sumant 1 al seu valor cada vegada que apareguin. Una vegada llegides totes les línies, s'escriurà el contingut del map en un nou fitxer.

5.3.4 Cerca d'alternatives des de les ciutats de CORDIS

Com s'ha vist en el capítol anterior, a partir del fitxer de *city-count.csv*, s'obtindrà el conjunt de tots els valors diferents que apareixen en el camp referent a la ciutat de les dades de CORDIS. El següent pas serà buscar per a cada un d'aquests valors si ja existeix en la base de dades de ciutats, ja sigui com a *City* o com a *Alt*. Si ja existeix, es donarà per vàlid aquest valor i s'actualitzarà el seu nombre d'aparicions en la base de dades de ciutats. Per contra, si aquest valor no existeix en la base de dades es procedirà a intentar trobar una ciutat o una alternativa similar que sí que hi sigui. Per això, serà necessari

Algorisme 5.4 Pseudocodi del càlcul d'aparicions d'una ciutat

```
1
2
3  function calcularAparicions(): void {
4
5      File file = participacions.csv;
6      File outputFile = city-count.csv;
7      CSVReader reader = new CSVReader(file);
8      CSVWriter writer = new CSVWriter(outputFile);
9      HashMap<String, Integer> map = new HashMap<String, Integer>();
10
11     String[] line;
12
13     while ((line = reader.readNext()!=null){
14         String nomCiutat = line[posCiutat] + "%" + line[posPais];
15
16         int aparicions = 1;
17         if (map.containsKey(nomCiutat)) {
18             aparicions = map.get(nomCiutat);
19             aparicions++;
20         }
21         map.put(nomCiutat, aparicions);
22     }
23
24     String writeLine[] = new String[3];
25     for (String city: map.keySet()){
26         writeLine[0] = city.substring(0,city.lastAppearance("%"));  //Nom ciutat
27         writeLine[1] = map.get(city);   //Aparicions
28         writeLine[2] = city.substring(city.lastAppearance("%") + 1);    //Nom país
29     }
30
31     reader.close();
32     writer.close();
33 }
```

disposar del país de manera obligatòria, ja que totes les comparacions es realitzaran només amb les ciutats d'un mateix país.

Cada vegada que es trobi una ciutat o una alternativa similar al valor que s'està buscant, es crearà dins de la base de dades un nou node *Alt*. A més, s'afegirà en un map de possibles relacions d'alternatives, on es guardarà l'identificador de l'alternativa creada i l'identificador de la ciutat o l'alternativa a la que s'assembla.

L'algorisme utilitzat en aquest procés es mostra en l'Algorisme 5.5.

Hi haurà dues maneres diferents de trobar relacions de similitud entre el valor que tenim i les ciutats/alternatives: les relacions de pertinença i les relacions de versemblança.

Relació de pertinença

Una forma de trobar aquestes relacions de similitud serà mirar si la possible ciutat provinent de les dades de CORDIS és una paraula o un conjunt de paraules contingut en una de les ciutats o de les alternatives de la base de dades.

Per exemple, si hi ha una ciutat amb el nom "Vilafranca del Penedès", i en les dades de CORDIS es troba el valor "Vilafranca", s'assumirà que "Vilafranca" pot ser una alternativa de "Vilafranca del Penedès". Per tant, es crearà una alternativa amb nom "Vilafranca" i s'afegirà en el map de relacions l'identificador del nou node de tipus *Alt* creat i l'identificador de la ciutat amb la que te aquesta relació de pertinença.

D'altra banda, també pot passar a l'inrevés. És a dir, si el valor conté alguna ciutat o alternativa de les ja existents també es considerarà que tenen una relació de pertinença i per tant el valor de la ciutat procedent de les dades de CORDIS passarà a ser una nova alternativa.

Un exemple seria el cas en què el valor que es pot trobar a les dades es "Barcelona Bellaterra". En aquest cas, es trobaria que tant "Barcelona" com "Bellaterra" són ciutats que estan contingudes en el valor que es compara. Així doncs, s'actualitzaria el map amb les noves relacions de similitud trobades.

Algorisme 5.5 Pseudocodi de la cerca d'alternatives

```
1  Graph graph = getDBGraph(); // Obtenim el graph de ciutats
2
3  function cercaAlternatives(): void {
4
5      File cityFile = city-count.csv;
6      File removedCityFile = removed-cities.csv;
7
8      CSVReader reader = new CSVReader(cityFile);
9      CSVWriter writer = new CSVWriter(removedCityFile);
10
11     String[] line;
12
13     while ((line = reader.readNext()!=null){
14
15         String nomCiutatPais = line[posCiutat] + "%" + line[posPais];
16         if (graph.findCity(nomCiutatPais)){
17             updateCount(nomCiutatPais, Integer.parseInt(line[posAparicions]));
18         }
19         else if (graph.findAlternative(nomCiutatPais){
20             updateCount(nomCiutatPais, Integer.parseInt(line[posAparicions]));
21         }
22         else{
23             if (!findSimilars(nomCiutatPais,Integer.parseInt(line[posAparicions]))
                   ) {
24                 writer.writeNext(line); //Escribim en el fitxer de ciutats a
                       eliminar
25             }
26         }
27     }
28     reader.close();
29 }
30
31 //Actualitza les aparicions d'una ciutat o alternativa
32 function updateCount(String ciutat, int aparicions): void{
33
34     int ap = graph.getAttribute(ciutat, atributAparicions);
35     ap = ap + aparicions;
36     graph.setAttribute(ciutat, atributAparicions, ap);
37
38 }
```

Relació de versemblança

Una altre opció serà intentar buscar en el conjunt de ciutats/alternatives existents aquella que més s'assembli al valor que tenim com a possible ciutat. Per fer-ho, s'utilitzarà el mètode de comparació de strings de *Levenshtein*. Per millorar l'eficiència, només es realitzaran les comparacions en les que la primera lletra del valor i la primera lletra de la ciutat o alternativa coincideixin. A mesura que es vagin fent les comparacions, s'anirà actualitzant quina és la ciutat o alternativa més similar al valor que tenim, de manera que com a màxim obtindrem 1 parella de similars final utilitzant la relació de versemblança.

En el cas de les comparacions amb alternatives, el resultat de la similitud tindrà una petita correcció, ja que en el cas que existissin una alternativa molt similar al valor que es compara i una ciutat que també té molta similitud però una mica menys que l'anterior, és preferible crear aquesta relació de similitud amb la ciutat.

Per exemple, si hi ha una ciutat amb nom "Barcelona" i una alternativa amb el nom "Barsalona" i s'està buscant la relació de versemblança de la paraula "Barcalona", serà preferible quedar-se directament amb la ciutat "Barcelona", encara que el resultat de la comparació sigui idèntic. És per això que s'introdueix una petita penalització al comparar amb alternatives.

Cal dir que s'establirà un llindar mínim de similitud, de manera que si la ciutat o alternativa més similar no ho és prou, no s'afegirà. En aquest cas, el valor que comparàvem provinent de les dades de CORDIS s'escriurà en un nou fitxer de valors a eliminar que s'utilitzarà més endavant .

L'algorisme utilitzat per a la cerca de similars es mostra de forma abreujada en l'Algorisme 5.6. En aquest cas, només es mostra la part referent a la cerca de similars entre les ciutats del graf. La cerca entre les alternatives es realitza de manera anàloga, amb el canvi que quan es troba una relació de similaritat amb una alternativa *Alt1*, el map de relacions s'actualitzarà amb totes les parelles formades per l'identificador de la nova alternativa i l'identificador de cada una de les ciutats de les quals *Alt1* ja estava relacionada per mitjà de l'aresta *isAlternative*. Conseqüentment, les aparicions de totes aquestes ciutats també s'actualitzaran.

Algorisme 5.6 Pseudocodi de la cerca de similars per ciutats

```
1  double llindar;
2  HashMap<String,List<String>> map = new HashMap<String,List<String>>();
3
4  function findSimilars(String nomCiutatPais, int aparicions): boolean {
5
6      String ciutat = nomCiutatPais.substring(0,nomCiutatPais.lastAppearance("%"));
           //Nom ciutat
7      String pais = nomCiutatPais.substring(nomCiutatPais.lastAppearance("%") + 1);
           //Nom pais
8      boolean similarTrobat = false;
9      double maxSim = 0;
10     String ciutatMaxSim = "";
11
12     //Cerca de similars en ciutats
13     Objects ciutatsGraph = graph.select(cityType, atributPais, op_eq, pais); //
           obtenim totes les ciutats del graph del pais
14
15     for (Object ciutat: ciutatsGraph) {
16         String ciutatNom = graph.getAttribute(atributCiutat, ciutat);
17
18         //Pertinença
19         if(ciutatNom.contains(ciutat) || ciutat.contains(ciutatNom)){
20             updateCount(nomCiutatPais, aparicions);
21             updateAltMap(ciutat, ciutatNom);
22             similarTrobat = true;
23             graph.newNode(altType, nomCiutatPais);
24         }
25
26         //Versemblança
27         if (ciutatNom.getCharAt(0).equals(ciutat.getCharAt(0))){
28             double sim = compareLevenshtein(ciutat, ciutatNom);
29             if (sim > maxSim){
30                 maxSim = sim;
31                 ciutatMaxSim = ciutatNom;
32             }
33         }
34     }
35
36     ciutatsGraph.close();
37
38     if (maxSim > llindar) {
39         updateCount(nomCiutatPais, aparicions);
40         updateAltMap(ciutat, ciutatMaxSim);
41         similarTrobat = true;
42         graph.newNode(altType, nomCiutatPais);
43     }
44
45     return similarTrobat;
46 }
47
48
49 function updateAltMap(String alternativa, String ciutatOriginal): void {
50
51     List l = new ArrayList();
52     if (map.containsKey(alternativa)) {
53         l = map.get(alternativa);
54     }
55     l.add(ciutatOriginal);
56     map.put(alternativa, l);
57 }
```

5.3.5 Selecció de les relacions de similitud

Després del procés de la cerca de relacions de similitud s'obté un conjunt de nodes *Alt* que s'han creat en la base de dades i un map indicant per a cada un d'aquests nous nodes un conjunt de ciutats de les quals pot ser alternativa. En aquest moment, el principal objectiu serà triar quines d'aquestes relacions es carreguen al graf que forma la base de dades.

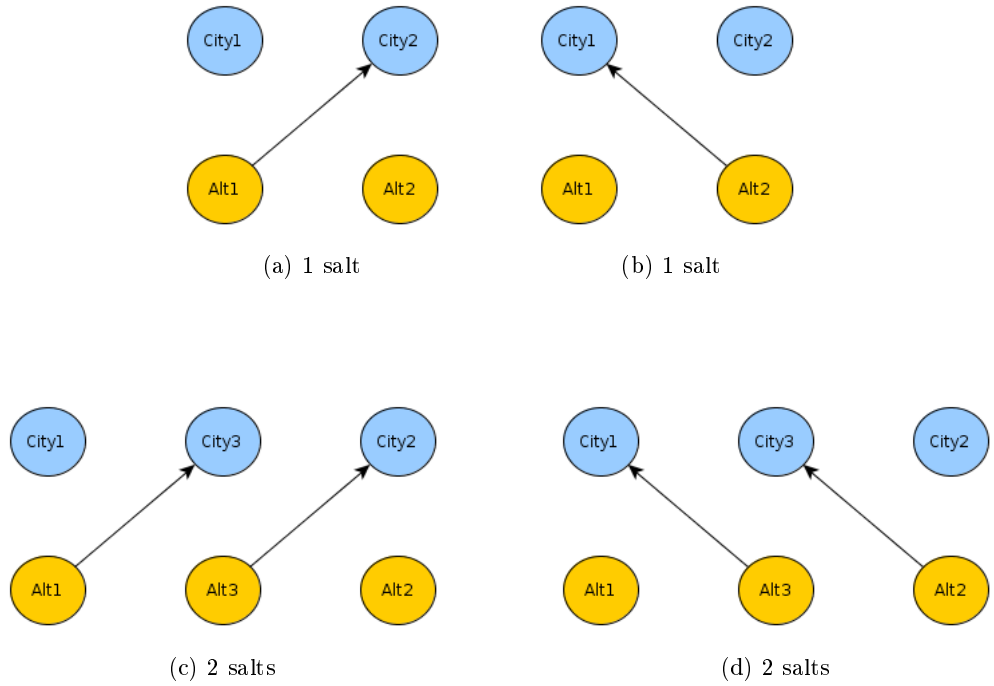
En primer lloc, s'intentarà seleccionar la millor relació de similitud de les de la llista. Per definir quina es la millor, se seguiran els següents criteris:

- Si el nom del nou node creat conté el nom d'algunes ciutats de la llista, la millor serà alguna d'aquelles ciutats.
  - En cas d'haver-hi varies ciutats contingudes, la millor serà la que tingui més paraules, ja que generalment serà la més específica.
  - En cas d'igual nombre de paraules, s'escollirà a la ciutat que tingui menys aparicions, seguint el criteri de triar la ciutat que tingui més especificitat. Per fer això, s'haurà de calcular el nombre d'aparicions per cada ciutat en les dades de CORDIS.
- Si el nom del nou node no conté cap ciutat, s'escollirà d'entre totes les alternatives possibles aquella que tingui un nombre d'aparicions més elevat. Això es realitza per tal de tenir una probabilitat més gran d'encertar quina és la ciutat corresponent.

Sempre es carregarà la relació entre el node creat i la ciutat que s'ha triat com a millor. De totes maneres, es realitzarà un procés de revisió per intentar esbrinar si hi ha altres ciutats "properes" en el graf que també puguin ser candidates a tenir com a alternativa el node creat.

El propi esquema del graf, fa possible que hi hagi ciutats i alternatives amb un mateix nom. Aleshores, s'acceptaran com a ciutats properes (i per tant es crearà una relació entre l'alternativa creada i la ciutat) aquelles que apareguin en la llista del map i tinguin un camí de 2 o menys salts entre la millor ciutat i l'alternativa de l'altre o a l'inrevés, és a dir, entre l'alternativa de la millor ciutat i l'altre ciutat.

En la Figura 5.8 es poden veure els casos en que considerem que dues ciutats són properes. En tots els casos, *Alt1* té el mateix nom que *City1*, *Alt2* que *City2* i *Alt3* que *City3*. Tal



(a) 1 salt (b) 1 salt

(c) 2 salts (d) 2 salts

Figura 5.8: Casos on City1 i City2 són considerades ciutats properes.

i com es pot veure, els casos *a* i *b* fan referència a ciutats a 1 salt de distància, mentre que en els casos *c* i *d*, les ciutats estaran a 2 salts de distància. Així doncs, si *City1* és la millor ciutat entre les de la llista i *City2* és una altre ciutat qualsevol que està en el map, i es compleix algun d'aquests casos, *City2* seria considerada una ciutat propera i per tant es crearia una nova aresta *isAlternative*.

5.3.6 Creació del fitxer de ciutats i alternatives úniques

Una vegada ja s'ha aconseguit tenir el graf complet amb totes les noves relacions *isAlternative*, serà necessari crear un fitxer que contingui tots els possibles valors vàlids que pot prendre el camp ciutat del fitxer de participacions. Per fer-ho, es farà una cerca en la

base de dades de totes les ciutats i totes les alternatives, i s'aniran escrivint en un nou fitxer.

5.3.7 Obtenció de ciutats a partir de la direcció

Un cop realitzat tot aquest procés, es realitzarà una neteja del camp ciutat del fitxer de participacions. S'eliminaran tots els valors que hi apareguin i que també estiguin en la llista de ciutats a eliminar (formada per aquells valors de ciutats que no hi seran en el graf, ja que són possibles ciutats per les que no s'han trobat similituds prou bones).

En el cas que aquest camp estigui buit, ja sigui per l'eliminació de valors que s'ha realitzat o simplement perquè a les dades estava el camp buit, es procedirà a intentar trobar la ciutat en l'adreça. Els passos que se seguiran són els següents:

- Realitzar transformacions en el camp adreça: eliminar tots els números i normalitzar-lo, eliminant accents i signes de puntuació.
- A partir d'un fitxer de països, buscar si apareix el país en l'adreça, guardar-lo en el camp corresponent si el valor era nul i eliminar-lo de l'adreça.
- Buscar si existeix alguna de les ciutats o de les alternatives del graf en l'adreça a partir del fitxer de ciutats/alternatives úniques creat després del procés de selecció de similituds. La ciutat o alternativa seleccionada serà aquella que tingui més paraules (serà la més específica) i que estigui més a prop del final de l'adreá, ja que aquestes solen començar amb el carrer, continuar amb la ciutat i acabar amb el país. D'aquesta manera, s'evita agafar noms de carrers com a ciutats.

Un exemple d'aquest últim punt es pot trobar en l'exemple següent:

Si es disposa d'una direcció amb el valor "C/Girona 25, Hospitalet del Infant, Espanya", es realitzaran les transformacions del primer punt. Així, quedarà una direcció amb el valor següent:

- c girona hospitalet del infant espanya.

A continuació es buscaran països dins de l'adreça i es trobarà "espanya", així que es posarà en el camp país si aquest es null i s'eliminarà de l'adreça, quedant aquesta de la següent manera:

- c girona hospitalet del infant.

En aquest moment es buscaran les ciutats del graf existents en aquesta adreça. Es trobaran tres ciutats o alternatives: "girona", "hospitalet" i "hospitalet del infant". En aquest cas, la ciutat seria "hospitalet del infant" que es la que té més paraules i per tant es considera la més concreta. Així doncs, aquest valor seria el que es posaria en el camp ciutat que no en tenia cap.

5.3.8 Estat final al acabar la deduplicació de ciutats

Un cop acabat tot el procés de deduplicació de resultats es disposa dels següents nous elements:

- Base de dades amb ciutats i alternatives, juntament amb les relacions de cada ciutat amb les seves possibles alternatives.
- Fitxer de ciutats/alternatives que existeixen en el graf.
- Fitxer de participacions on el seu camp ciutat disposarà d'un valor nul o d'una ciutat o alternativa que estigui en la base de dades de ciutats.

Aquests tres elements seran necessaris en la deduplicació d'institucions tal i com es veurà més endavant.

5.4 Deduplicació d'institucions per nom

5.4.1 Descripció general

Tant la deduplicació de ciutats com la de regions estan enfocades a permetre una millor deduplicació d'institucions, que tindrà dos enfocaments diferents a l'hora de fer els blocs: a partir dels noms i a partir de les sigles. En la deduplicació per nom, serà necessari normalitzar els noms de les institucions per facilitar-ne les comparacions. Una vegada normalitzats, les institucions s'ordenaran per nom. Aquest fitxer ordenat, juntament amb altres, seran la base a partir de la qual buscar els duplicats de les institucions mitjançant

el *framework* de *Dexdup*. L'esquema d'aquesta deduplicació es pot trobar en la Figura 5.9.

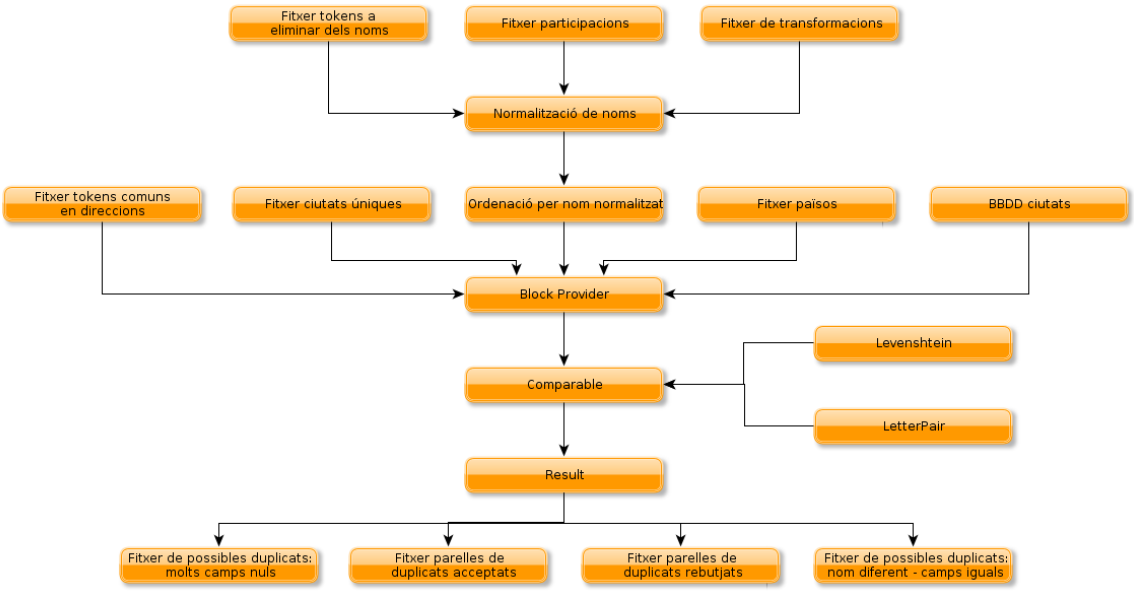


Figura 5.9: Procés general de la deduplicació d'institucions per nom.

5.4.2 Normalització de noms

En moltes ocasions, els noms de les institucions estan escrits en diferents idiomes (*universitat*/*university*) i/o contenen paraules que no aporten informació per a la deduplicació com podrien ser articles, preposicions o paraules concretes que per ser massa comunes en els noms, no són rellevants per a la deduplicació ("societat anònima" o "societat limitada" en serien exemples). És per això que es realitza un procés previ de normalització de noms.

Aquest procés es divideix en dues parts: l'eliminació de tokens irrellevants i les substitucions de paraules. Una vegada realitzat, el fitxer de participacions resultant s'ordenarà per aquest nom normalitzat.

Algorisme 5.7 Pseudocodi de l'eliminació de tokens irrellevants

```
function eliminaTokens(): void{

    File f = remove-tokens.csv
    File fPart = participations.csv
    File outputFile = participations-name-normalized.csv

    CSVReader removeReader = new CSVReader(f);
    CSVReader partReader = new CSVReader(f);
    CSVWriter writer = new CSVWriter(outputFile);

    String[] line = removeReader.readNext();
    String tokenList = "";

    if (line!= null) {
        tokenList = "\\b(" + line[0];
    }
    while ((line=removeReader.readNext()) != null) {
        tokenList = tokenList.concat("|" + line[0]);
    }
    tokenList = tokenList.concat(")\\b");

    while ((line = partReader.readNext() )!=null){
        String[] writeLine = new String[line.length + 1];
        for (int i = 0; i<line.length; i++) {
            writeLine[i] = line[i];
        }
    }

    writeLine[line.length] = writeLine[line.length].replaceAll(tokenList, "");
    writer.writeNext(writeLine);
}
```

Eliminació de tokens irrellevants

Es parteix d'un fitxer de tokens que es consideren irrellevants. Aquest fitxer ha estat creat prèviament tenint en compte els tokens més comuns en el camp del nom de les institucions. A partir d'aquesta llista de tokens més freqüents, s'ha realitzat una revisió manual per afegir tokens similars (encara que no es repeteixin tantes vegades) o eliminar tokens que tot i ser molt repetits poden aportar informació per a la deduplicació.

L'Algoritme 5.7 mostra com es realitza aquest procés: en primer lloc es recorrerà el fitxer amb els tokens a eliminar i s'aniran guardant en una cadena. Posteriorment, s'anirà recorrent el fitxer de participacions línia a línia i s'anirà afegint una última columna amb el nom de la institució però fent un *replaceAll* per elimnar tots els tokens irrellevants.

Substitucions de paraules

S'utilitza un fitxer de transformacions creat manualment, amb la forma "paraula a substituir|nova paraula". S'usa principalment per fer que l'idioma amb que estigui escrit el nom no afecti tan en la deduplicació. Per posar un exemple, es troben escrites més de 20 maneres diferents per referir-se al terme "universitat" en els noms de les institucions. Això provocaria moltes diferències en les comparacions de noms, i el simple fet de traduir alguns termes dels més comuns, ajuda molt en l'obtenció de millors resultats.

De la mateixa manera que abans, s'aniran cercant els diferents tokens, però en comptes de eliminar-los, se substituiran pel valor indicat en el fitxer de transformacions, que s'haurà carregat a l'inici.

5.4.3 Creació de blocs

És en aquest punt on es comença a fer ús de *Dexdup*, el *framework* de deduplicació creat. Com s'ha explicat, no es pot fer una comparació entre totes les possibles parelles d'institucions, ja que serien necessàries $2 * 10^{11}$ comparacions aproximadament, la qual cosa tindria un cost computacional inacceptable. És per això, que en aquesta deduplicació s'utilitzarà un mètode de *blocking*, concretament el mètode de *sliding window*.

Input

Al *BlockProvider*, arribaran 5 elements a partir dels quals realitzar els blocs:

- Fitxer de participacions ordenat segons el nom normalitzat.
- Fitxer de ciutats úniques.
- Fitxer de països únics.
- Fitxer de tokens a eliminar de les adreces.
- Base de dades amb les ciutats.

Ja s'ha explicat la procedència de tots aquests elements excepte el fitxer de tokens a eliminar en les adreces. Aquest fitxer contindrà un conjunt de paraules freqüents en les adreces que no ens permetran realitzar bones comparacions. D'aquesta manera, s'eliminaran totes les paraules de l'estil *carrer, rue, calle, avenue, street...* i les seves abreviatures, per tal que la informació restant en la direcció sigui més representativa. De la mateixa manera que amb l'anterior fitxer utilitzat per eliminar tokens, aquest s'ha creat a partir d'un llistat dels tokens més freqüents en l'adreça revisat manualment a posteriori.

Implementació funcions *hasNext()* i *next()*

Dexdup obté els blocs a partir de la interfície *BlockIterator,* que té dues funcions. La funció *hasNext()* ha de retornar true si encara hi ha més blocs, mentre que la funció *next()* retorna el bloc pròpiament dit.

La primera vegada en executar *hasNext(),* es realitza una inicialització. Els fitxers de ciutats úniques, països i tokens a eliminar es carreguen a memòria i s'inicialitzen els lectors. A partir d'aquí, cada vegada que es cridi la funció, aquesta llegirà la següent línia del fitxer de institucions i si la línia no és nul·la, retornarà cert.

D'altra banda, la funció *next()* farà el següent: si el bloc a retornar està buit, l'omplirà amb les següents N institucions fins que en el bloc hi hagi M institucions amb noms diferents, on M serà un valor modificable que estarà inicialitzat a 50 per defecte i N serà un nombre variable d'institucions en funció de les institucions amb noms iguals que apareguin. En canvi, si el bloc està ple, s'eliminarà la primera institució del bloc. Si la següent institució del bloc té un nom diferent al de la institució eliminada, es tornaran a afegir institucions al bloc fins a tenir les 50 institucions diferents, si no, es retornarà el bloc directament. El pseudocodi d'aquesta part del *BlockIterator* es pot veure en l'Algorisme 5.8, on M està representada en la variable *diferentElements*.

Al afegir institucions en el bloc, ja s'introduiran nous valors calculats per tal de fer més àgil les posteriors comparacions. Així doncs, a més del nom, la ciutat, el país, la regió...també s'afegiran valors com el nom normalitzat, el nom sense espais, el nom separat per paraules, l'adreça normalitzada i una llista d'alternatives de la ciutat.

Per realitzar la normalització de l'adreça, s'eliminaran els tokens comuns carregats en la inicialització i es buscarà en la mateixa adreça si hi ha el nom d'algun país o de la ciutat

Algorisme 5.8 Pseudocodi de la funció *next()* del BlockIterator

```
1
2  ArrayList<Object> block = new ArrayList<Object>();
3  int diferents = 0;  //Nombre d'elements amb noms diferents existents en el block
       actual
4  int diferentElements = 50   //Nombre d'elements amb noms diferents que es vol
       tenir en cada block.
5  CSVReader readerNext(); //Reader del fitxer d'institucions inicialitzat en el
       hasNext().
6  String firstName; //nom de la primera institució del block actual
7  String lastName;  //nom de l'última institució del block actual
8
9  function next(): Collection<Object> {
10
11     if (!block.isEmpty()) {
12         Institution primer = (Institution) block.get(0);
13         if (!primer.getName().equals(firstName)) {
14             firstName = primer.getFilteredName();
15             diferents--;
16         }
17         block.remove(0);
18     }
19
20     boolean end = false;
21     while ((diferents < diferentElements) && !end) {
22
23         String[] line;
24
25         if ((line=readerNext.readNext())!=null){
26             Institution inst = new Institution();
27             inst.setAttributes(line); // es posen els attributs segons
                   corresponents segons la línia del fitxer de participacions
28             block.add(inst);
29             if (inst.getName().equals(lastName)) {
30                 lastName = inst.getName();
31                 diferents++;                        }
32             }
33         }
34         else{
35             end = true;
36         }
37     }
38
39     return block;
40 }
```

a la que pertany la institució. Si es troba un país o una ciutat també s'eliminarà. Això es realitza perquè en les comparacions seria redundant tenir en compte la ciutat o el país en dos camps a la vegada.

D'altra banda, la llista d'alternatives s'obtindrà a partir de la base de dades de ciutats que disposem. Es buscaran totes les possibles alternatives al nom d'una ciutat i s'afegiran els seus identificadors a una llista.

5.4.4 Comparacions

Dexdup està implementat de manera que el *BlockProvider* obté una sèrie de blocs d'institucions. Per a cada bloc es realitzaran les comparacions, utilitzant la metodologia *One-Vs-All*, és a dir, es compararà el primer element amb tots els altres elements del bloc. Aquestes comparacions s'implementaran en la classe *UsingComparablesInstitution*, que estendrà de *DecisionAware*, per tal de permetre afegir missatges sobre l'estat de la comparació. El procés que se seguirà a l'hora de realitzar una comparació entre dues institucions serà el descrit en la figura 5.10.

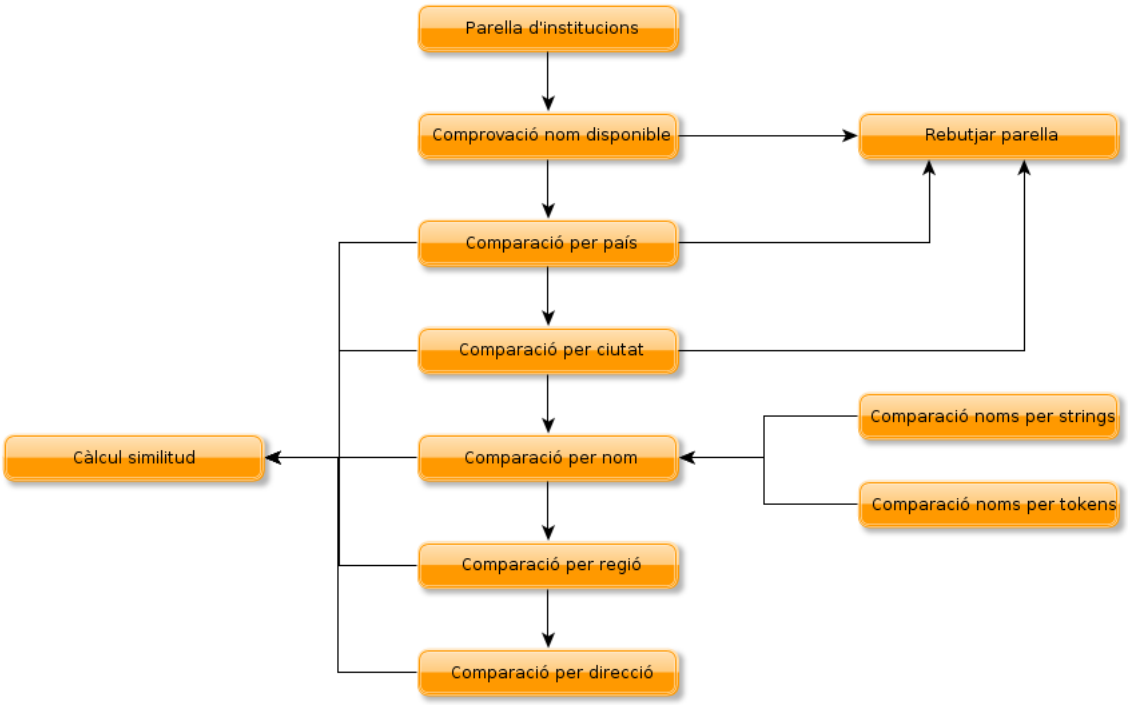


Figura 5.10: Procés de comparació d'institucions per noms.

Així doncs, per a cada una de les parelles d'institucions a comparar es realitzarà el procés descrit a continuació, que s'anirà representant en pseudocodi per parts i de manera abreujada per tal de facilitar-ne la comprensió.

En primer lloc, s'inicien les variables on es guardaran els resultats de les comparacions de ciutats, països o regions, a més de la variable *decision*, que permetrà afegir un missatge al resultat. Un cop fet això, es voldran rebutjar directament comparacions amb noms incorrectes. Per això, si com a mínim un dels noms de les institucions a comparar és "*not available*", es rebutjarà la comparació directament (Algorisme 5.9).

Un criteri que s'estableix d'inici en aquesta deduplicació és el de deduplicar a nivell de ciutats. És a dir, dues institucions de dues ciutats diferents mai podran ser considerades duplicats. És per això que en primer lloc es compararan els països de les institucions, i tot seguit, es compararan les ciutats de les institucions, sempre que aquestes existeixin.

Algorisme 5.9 Pseudocodi del rebuig de comparacions per noms no disponibles

```
function compare(Institution inst1, Institution inst2): double {
    double result = 0;
    double resCountry;
    double resCity;
    double resName;
    double resRegion;
    double resAddress;
    Decision decision = new Decision(); //Permet retornar un missatge amb el
        resultat

    //Si algun nom no està disponible, retornem 0
    if (inst1.getName().equals("not available") || inst2.getName().equals("not
        available")){
        return 0;
    }
}
```

Per a realitzar la comparació dels països, s'utilitzarà el seu codi, obtenint valor 1 si són iguals. En cas que els països no coincideixin, es rebutjarà directament la comparació. D'aquesta manera, no caldrà fer tots els altres passos del càlcul de similituds entre dos institucions, que és la part de la deduplicació que necessita un temps d'execució més elevat per a completar-se. Aquesta part del codi queda reflectida en l'Algorisme 5.10.

Algorisme 5.10 Pseudocodi de la comparació de països

```
//Comparació països
if (inst1.getCountryCode() == null || inst2.getCountryCode == null){
    if(inst1.getCountryCode() == inst2.getCountryCode()){
        resCountry = 1;
    }
    else{
        return 0;
    }
}
```

Així doncs, si els països de les institucions coincideixen o n'hi ha alguna que no té disponible aquest camp, es procedirà a realitzar la comparació de ciutats. En aquest cas, no només es compararan les ciutats sinó també les seves alternatives. Així, si no hi ha cap possible alternativa de la ciutat de la primera institució que també sigui alternativa de la ciutat de la segona institució, es rebutjarà la comparació. Si pel contrari es troba

una coincidència, es posarà 1 com a valor de la comparació de ciutats, tal i com es mostra en l'Algorisme 5.11.

Algorisme 5.11 Pseudocodi de la comparació de ciutats

```
1  //Comparació ciutats
2  if (inst1.getCity() == null || inst2.getCity == null){
3      List altList1 = inst1.getAlternatives(); //retorna una llista de totes les
           ciutats possibles de la institució
4      List altList2 = inst2.getAlternatives();
5
6      for (String ciutat1: altList1) {
7          for (String ciutat2: altList2) {
8              if (ciutat1.equals(ciutat2)){
9                  resCity = 1;
10                 break;
11             }
12         }
13     }
14
15     if (resCity == 0) {
16         return 0;
17     }
18 }
```

A continuació se seguirà comparant amb el següent criteri que ens permet rebutjar directament alguna comparació, el nom de la institució. Realitzar una comparació per noms amb aquestes dades és força complicat, ja que una mateixa institució pot tenir noms molt diferents, ja sigui perquè el nom d'una institució té un nivell de concreció més elevat, perquè està en un altre idioma, perquè l'ordre de les paraules és diferent...

Per tal de solucionar aquesta problemàtica, es realitzaran comparacions per nom seguint diferents criteris i s'escollirà com a valor de la similitud de noms el valor màxim de totes aquestes comparacions.

En concret, les primeres comparacions que es realitzaran seran comparacions de strings. Se'n realitzaran a partir del nom normalitzat (sense accents, sense signes de puntuació, sense algunes paraules habituals com articles o preposicions...) i a partir del mateix nom normalitzat però eliminant els espais. Per a cada un d'aquests casos, la comparació serà realitzada per uns comparadors que implementen dos algorismes de comparació de strings com són *Levenshtein* i *LetterPair*. Si una comparació coincideix completament, ja no se'n realitzaran més. En cas contrari, es guardarà com a resultat el valor màxim d'entre totes

Algorisme 5.12 Pseudocodi de la comparació de noms

```
1  //Comparació noms
2  resName = this.getComparatorsMap().get("Levenshtein").compare(inst1.getName(),
       inst2.getname());
3  resName = Math.max(resName, this.getComparatorsMap().get("LetterPair").compare(
       inst1.getName(), inst2.getname()));
4  resName = Math.max(resName, this.getComparatorsMap().get("Levenshtein").compare(
       inst1.getNameNormalized(), inst2.getNameNormalized()));
5  resName = Math.max(resName, this.getComparatorsMap().get("LetterPair").compare(
       inst1.getNameNormalized(), inst2.getNameNormalized()));
```

aquestes comparacions, ja que aquest serà el valor que es tindrà en compte en el càlcul final de similitud entre les dues institucions. Això queda reflectit en l'Algorisme 5.12.

Sempre que cap d'aquestes comparacions hagi donat com a resultat un valor màxim, es procedirà a realitzar la comparació de noms d'institucions per tokens. En moltes ocasions en el nom d'una institució es troba inclosa informació addicional que fa que casos com "Universitat Politècnica Catalunya" i "Universitat Politècnica Catalunya - Departament d'arquitectura de computadors" tinguin resultats molt baixos en les comparacions per nom tot i ser la mateixa entitat. Per tal de sol·lucionar-ho s'utilitzarà aquesta comparació de noms d'institucions per tokens.

Aquesta comparació per tokens tindrà en compte si el nom d'una institució està inclòs en el nom de l'altre. Tot i així, hi haurà institucions amb noms de poques paraules que seran continguts per altres noms d'institucions amb més facilitat que altres noms d'institucions més llargs. Un cas a part, serien les institucions que tenen noms d'una sola paraula, que poden ser continguts per altres noms d'una manera molt fàcil. És per això, que es tindrà en compte el nombre de tokens de cada nom a l'hora de valorar aquesta similitud.

Així doncs, si els noms de les dues institucions tenen tres o més tokens i un dels noms està contingut en l'altre, considerarem que els noms són iguals, donant el màxim valor possible a la comparació (1.0). En el cas que els noms tinguin dos o més tokens i un estigui contingut en l'altre, es posarà com a resultat de la similitud de noms un 0.95. Per últim, si hi ha una institució amb un nom de només un token però que està contingut en l'altre nom d'institució, es marcarà per tractar aquest cas més endavant d'una manera especial, ja que no es pot donar un valor de la comparació molt elevat però tampoc es vol descartar que siguin duplicats.

En l'exemple anterior entre "Universitat Politècnica Catalunya" i "Universitat Politècnica Catalunya - Departament d'arquitectura de computadors", el resultat de la comparació seria 1, ja que els dos noms tenen més de tres tokens i el primer nom està contingut en el segon.

Addicionalment, també s'introduirà una nova comparació en el cas d'institucions de més de 3 paraules amb alguna variació en l'ordre de les mateixes, per tal de cobrir més possibilitats. En concret, s'invertirà l'ordre del segon i tercer token, ja que els primers tokens acostumen a ser els més descriptius i el primer és més probable que coincideixi per la manera en que es realitzen els blocs.

Una vegada s'hagin realitzat totes aquestes comparacions, es buscarà el valor màxim entre els resultats de les comparacions per tokens i per nom. Aquesta comparació per tokenització es mostra en l'Algorisme 5.13.

Algorisme 5.13 Pseudocodi de la comparació per tokenització

```
if (resName < 1) {
    //Comparació per tokens
    double containsRes = 0;
    if (inst1.getTokenizedName().length > 3 && inst2.getTokenizedName().length >
        3){
        boolean cont = contains(inst1,inst2); // retorna cert si inst1 conté inst
            2 o a l'inrevés
        if (cont) {
            containsRes = 1;
        }
    }
    else if(inst1.getTokenizedName().length > 2 && inst2.getTokenizedName().length
         > 2){
        boolean cont = contains(inst1,inst2); // retorna cert si inst1 conté inst
            2 o a l'inrevés
        if (cont) {
            containsRes = 0.95;
        }
    }
    else{
        boolean cont = contains(inst1,inst2); // retorna cert si inst1 conté inst
            2 o a l'inrevés
        if (cont) {
            containsRes = -1;
        }
    }
    resName = Math.max(resName,containsRes);
}
```

Sempre que tinguem activada l'opció *cutRejected*, directament es rebutjaran com a duplicats aquelles parelles que tinguin una similitud de noms inferior a 0.35 i que no estiguin dins del cas especial de tenir un únic token que a més estigui contingut en l'altre nom. Això evita realitzar càlculs innecessaris per institucions que no tenen noms que s'assemblen prou, millorant la velocitat d'execució del codi. Aquesta part es mostra en el pseudocodi de l'Algorisme 5.14.

Algorisme 5.14 Pseudocodi del *cutRejected*

```
1 //retornem 0 si tenen una similitud de nom molt baixa
2 if (cutRejected && containsRes!=-1) {
3     if (resName < 0.35) {
4         return 0.0;
5     }
6 }
```

Tot seguit, es realitzaran diverses comparacions menors que s'aniran guardant en valors independents per tal d'utilitzar-los al final del codi. Aquestes comparacions inclouran regions i adreces. Si algun d'aquests camps està buit o és nul, s'indicarà en un comptador de camps nuls i no es tindrà en compte aquell camp a l'hora de realitzar el càlcul final de la comparació d'institucions.

Pel que fa a les regions, es tindrà en compte tant el codi com la descripció de la regió. El codi de regió segueix l'estandard *NUTS* [18], que funciona a nivell europeu. Es tracta d'un codi jeràrquic per tal de donar uniformitat a les informacions estadístiques regionals a Europa, i que està format per dues lletres i diferents xifres. Les lletres indiquen el país, i les xifres defineixen les unitats administratives locals d'aquest país. En el cas d'Espanya, aquestes unitats administratives coincideixen amb una unió de varies comunitats autònomes veïnes en un primer nivell (*NUTS-1*). Segons hi hagi més xifres, el codi serà més concret, definint comunitats autònomes (*NUTS-2*), províncies (*NUTS-3*), i altres unitats més locals com podrien ser les comarques (*NUTS-4 i NUTS-5*). El fet de seguir aquesta nomenclatura a l'hora de mostrar el codi de la regió farà possible el fet de saber si una regió és una unitat territorial inclosa en una altre regió, ja que en aquest cas, les dues regions tindran la mateixa part inicial del codi. En la Figura 5.11 es pot veure la distribució de les regions amb codi *NUTS-2*, que són els codis que apareixen amb més freqüència a les dades.

Figura 5.11: Distribució de regions segons el codi *NUTS-2*.

Per tant, es donarà valor 1 a la comparació de regions si el codi o la descripció de la regió és igual, mentre que en els casos en els que el codi d'una regió estigui inclòs en l'altre es donarà un valor de 0.9, ja que una regió serà una subregió de l'altre.

Per les direccions, es tindrà en compte si una està inclosa en l'altre (es donaria valor 1 a la comparació). En cas contrari, el valor serà el resultat de la comparació de partícules realitzat amb el comparador *LetterPair*. El codi que mostra aquests càlculs és el de l'Algorisme 5.15.

En aquest punt, tindrem els valors de les següents comparacions:

- *resCountry*: resultat de la comparació de països (nul o 1). Mai serà 0, ja que ja hauríem rebutjat la comparació.

- *resName*: màxim de la comparació per noms ([0-1]). Mai serà nul, ja que ja hauríem rebutjat la comparació.

Algorisme 5.15 Pseudocodi de la comparació de regions i de direccions

```
1  //Comparació regions
2  if (inst1.getRegionCode() == null || inst2.getRegionCode() == null){
3      if(inst1.getRegionCode() == inst2.getRegionCode()){
4          resRegion = 1;
5      }
6      else if (containsRegionCode(inst1.getRegionCode(),inst2.getRegionCode())){
7          resRegion = 0.9;
8      }
9      else{
10         return 0;
11     }
12 }
13 else if (inst1.getRegionDescription() == null || inst2.getRegionDescription() ==
       null){
14     if(inst1.getRegionDescription().equals(inst2.getRegionDescription())){
15         resRegion = 1;
16     }
17     else{
18         return 0;
19     }
20 }
21
22 //Comparació direcció
23 if (inst1.getAddress() == null || inst2.getAddress() == null){
24     if (containsAddress(inst1.getAddress(),inst2.getAddress())){
25         resAddress = 1;
26     }
27     else{
28         resAddress = this.getComparatorsMap().get("LetterPair").compare(inst1.
               getAddress(), inst2.getAddress());
29     }
30 }
```

- *resCity*: resultat de la comparació de ciutats (nul o 1). Mai serà 0, ja que ja hauríem rebutjat la comparació.
- *resRegion*: resultat de la comparació de regions (nul o [0-1]).
- *resAddress*: resultat de la comparació d'adreces (nul o [0-1]).

Dins del càlcul per a establir el valor de la comparació entre les dues institucions, el pes de la comparació *resAddress*, sempre i quan aquest camp no sigui nul, serà fix (0.06), mentre que el pes dels altres camps serà variable. Això es degut a que les adreces són molt diferents entre sí i no es vol donar tanta importància en aquest camp.

Els camps *resCity*, *resCountry* i *resRegion*, es repartiran a parts iguals un pes de 0.24 si existeix el camp *resAddress*, o un 0.3 en cas contrari. Si algun d'aquests camps és nul, la resta es repartiran el mateix percentatge.

El camp *resNom* tindrà un pes de 0.7 sempre que existeixi algun valor entre *resCity*, *resCountry* i *resRegion*, 0.94 si només existeix el camp *resAddress* i 1 si no hi ha cap altre camp diferent de nul. La manera com es calcula el resultat final es mostra en l'Algorisme 5.16.

Per exemple, si tenim els següents valors:

- *resName*: 0.95.
- *resCountry*: 1.
- *resCity*: 1.
- *resRegion*: Nul.
- *resAdress*: 0.4.

El càlcul per obtenir el resultat seria:

$$0.95*0.7 + 1*(0.24/2) + 1*(0.24/2) + 0.4*0.06 = 0.665 + 0.12 + 0.12 + 0.024 = 0.929.$$

Algorisme 5.16 Pseudocodi del procés per calcular el resultat final

```
double resNotNull = countNotNulls(resCountry, resCity, resRegion, resAddress); //
    nombre de resultats no nulls

double addressPerc = 0.06;
double namePerc = 0.7;
boolean address = false;

if(resAddress != null){
    result += addressPerc * res6;
    address = true;
}
if(resCountry != null){
    if (address) {
        result += ((1-namePerc - addressPerc)/resNotNull) * resCountry;
    }
    else {
        result += ((1-namePerc)/resNotNull) * resCountry;
    }
}
if(resRegion != null){
    if (address) {
        result += ((1-namePerc - addressPerc)/resNotNull) * resRegion;
    }
    else {
        result += ((1-namePerc)/resNotNull) * resRegion;
    }
}
if(resCity != null){
    if (address) {
        result += ((1-namePerc - addressPerc)/resNotNull) * resCity;
    }
    else {
        result += ((1-namePerc)/resNotNull) * resCity;
    }
}

if (resNotNull==0) {
    namePerc = 1;
}
else if (resNotNull==1 && res6!=null){
    namePerc = 1-addressPerc;
}

double resCamps = result;
result += resName*namePerc;
```

Tot i que el camp *resName* és molt important en la comparació, succeeix que a vegades dues institucions són pràcticament idèntiques en tots els camps, però per diversos factors tenen noms massa diferents com per arribar al llindar mínim. És per això que en el cas que el resultat final estigui per sota del *threshold* determinat per acceptar (predefinit a 0.9), es repescaran aquelles comparacions que no tinguin cap camp nul i on el valor ponderat dels camps sigui com a mínim un 95% del màxim possible. A més també s'estableixen algunes restriccions en quan a la longitud de les adreces per tal de no donar per bones les que tinguin de pocs caràcters. Una altra condició important és la de tenir un mínim de 35% de similitud en el nom (no es vol ajuntar empreses totalment diferents d'un mateix edifici) o que una una de les institucions tingui un nom de només un token que estigui inclòs en el nom de l'altre institució. En el cas que tot això es compleixi, s'afegirà un missatge al resultat indicant que es tracta d'un cas especial susceptible de ser acceptat com a duplicat i es posarà el valor llindar com a valor del resultat.

Finalment, si el valor del camp *resCity* és nul, també afegirem un missatge al resultat per tal de tractar de manera diferent les institucions sense ciutat posteriorment. Aquestes dues situacions especials es mostren en l'Algorisme 5.17.

Algorisme 5.17 Pseudocodi de casos especials

```
1  if(result<threshold && resNotNull == 4 && (res>0.35 || containsRes==-1)&& resCamps
       >=(1-namePerc)*0.90 && inst1.getStreet().length()>7 && inst2.getStreet().
       length()>7){
2      result = 0.9;
3      decision.setDecisionType(DecisionType.UNKNOWN);
4      decision.setDescription("Equal␣fields␣but␣diferent␣name");
5  }
6
7  if (resCity == null) {
8      decision.setDecisionType(DecisionType.UNKNOWN);
9      decision.setDescription("An␣institution's␣city␣field␣is␣null");
10 }
11
12 return result;
```

5.4.5 Resultats

Una vegada s'ha obtingut un valor en la comparació d'institucions, aquesta informació arriba fins a la classe *ResultInstitutionPrinter* que implementa *Result*. El que farà aquesta

classe serà escriure els fitxers amb els resultats. En concret, es poden arribar a crear quatre fitxers, dels quals un és opcional.

En primer lloc, totes aquelles comparacions que tinguin un estat indefinit i un missatge indicant que tenen noms diferents però camps molt semblants, s'escriuran al fitxer *duplicats-nom-institutions.csv*.

D'altre banda, les comparacions que tinguin un estat indefinit i un missatge indicant que el camp ciutat és nul i que a més superin el valor llindar, s'escriuran en el fitxer *cityNulls-institutions.csv*.

Les comparacions que no entren en cap dels casos anteriors i tenen un valor per sobre del llindar, s'escriuran en el fitxer *duplicats-confirmats-institutions.csv*.

Finalment, les comparacions que tenen un valor inferior al llindar, si l'opció *writeRejected* està activada, seran escrites en un fitxer anomenat *duplicats-rebutjats-institutions.csv*. Aquest es l'únic fitxer no necessari dels que es creen, de manera que per agilitzar el procés de deduplicació, l'opció *writeRejected* està desactivada per defecte.

L'algorisme creat pel tractament de resultats es troba en l'Algorisme 5.18, tot i que en una versió simplificada. Per exemple, l'ordre dels identificadors de les institucions serà rellevant, tot i que en aquest pseudocodi no s'ha tractat per tal d'obtenir una versió més senzilla d'entendre.

Algorisme 5.18 Pseudocodi de tractament de resultats

```
File outputFile;      //Fitxer de duplicats de sortida
File rejectedFile;    //Fitxer de duplicats rebutjats
File cityNullsFile;   //Fitxer de duplicats amb ciutat nula
File dupNameFile;     //Fitxer de duplicats amb nom nul

CSVWriter outputWriter;
CSVWriter rejectedWriter;
CSVWriter cityNullsWriter;
CSVWriter dupNameWriter;

boolean init = false;
boolean writeRejected;

function addResult(Object element1, Object element2, Double sim, Decision decision
    ): void {
    if (!init){
        iniciarWriters();
        init = true;
    }

    if (decision.getDecisionType() == DecisionType.UNKNOWN) {
        if (sim >= threshold) {
            if (decision.getDescription().equals("Equal fields but diferent name")
                ) {
                dupNameWriter.writeNext(getResultLine(element1, element2, sim,
                    decision));
            }
            else {
                cityNullsWriter.writeNext(getResultLine(element2, element1, sim,
                    decision));
            }
        }
        else if (writeRejected ) {
            rejectedWriter.writeNext(getResultLine(element1, element2, sim,
                decision));
        }
    }
    else if (sim >= threshold) {
        writer.writeNext(getResultLine(element1, element2, sim, decision));
    }
    else if (writeRejected ) {
        rejectedWriter.writeNext(getResultLine(element1, element2, sim, decision))
            ;
    }
}

//Funció que crea un String[] amb els camps adequats
function getResultLine(Object element1, Object element2, Double sim, Decision
    decision): String[]{}
```

5.4.6 Fitxer de configuració

El fitxer de configuració de Dexdup utitlitzat en el procés de deduplicació d'institucions per nom serà el mostrat en el següent codi:

```
<config>

  <deduplication name="institutions" active="true">

    <blockProvider bean="BlockIteratorInst">
      <property name="file" value="project-participations-name-sorted.csv"/>
      <property name="cityFile" value="city-uniq.csv"/>
      <property name="tokensFile" value="token-address-list.csv"/>
      <property name="countryFile" value="countries.csv"/>
      <property name="separator" value="|"/>
      <property name="quoteChar" value="\0"/>
      <property name="diferentElements" value="50"/>
    </blockProvider>

    <comparisonMethod name="LevenshteinDist" bean="UsingComparablesInstitution">
      <property name="nullWeight" value="0.5"/>
      <property name="cutRejected" value="true"/>
      <property name="threshold" value="0.9"/>
      <propertyComparator name="Levenshtein" comparator="
          LevenshteinDistanceComparable"/>
      <propertyComparator name="LetterPair" comparator="LetterPairComparable"/>
      <propertyComparator name="EqualComparator" comparator="EqualComparable">
        <property name="nullWeight" value="0.3"/>
      </propertyComparator>
    </comparisonMethod>

    <result bean="ResultInstitutionPrinter">
      <property name="outputFile" value="duplicats-confirmats-institutions.csv"/>
      <property name="rejectedFile" value="duplicats-rebutjats-institutions.csv"/>
      <property name="cityNullsFile" value="cityNulls-institutions.csv"/>
      <property name="dupNameFile" value="duplicats-nom-institutions.csv"/>
      <property name="separator" value="|"/>
      <property name="quoteChar" value="\0"/>
      <property name="threshold" value="0.9"/>
      <property name="writeRejected" value="false"/>
    </result>

  </deduplication>

</config>
```

5.5 Deduplicació d'institucions per sigles

5.5.1 Descripció general

Un dels problemes importants amb aquest conjunt de dades és la seva gran dimensió. En el cas de la deduplicació d'institucions és particularment problemàtic. Es pot donar el cas en què, tot i que dues institucions siguin pràcticament iguals, pel fet que el seu nom comenci amb una lletra diferent no s'arribin a comparar mai. Per solucionar aquest problema, a més d'una deduplicació d'institucions per noms, també es realitzarà una deduplicació per sigles o acrònims. La idea general d'aquesta serà buscar tots els possibles acrònims d'una institució (a partir de diferents camps com el nom o la pàgina web) per tal de comparar posteriorment totes les institucions que poden tenir un mateix acrònim entre elles. Així doncs, el mètode que se seguirà a l'hora de fer els blocs serà el de *Standard Blocking*.

5.5.2 Cerca de possibles sigles

La cerca de sigles es realitzarà en els camps referents al nom i a la pàgina web de la institució, com ja s'ha comentat.

Pel que fa a la pàgina web, com a primer pas sempre s'eliminaran tots els caràcters abans del primer punt i després de l'últim. Un cop realitzat això, es consideraran possibles acrònims d'una institució:

- Tot allò que quedi, sempre i quan no hi hagi més punts o guions ("www.upc.edu" –> "UPC").
- Les paraules que s'obtinguin al separar el que queda del string de la pàgina web per punts ("www.fib.upc.edu" –> "FIB" i "UPC").
- Les paraules que s'obtinguin al separar el que queda del string de la pàgina web per guions ("www.dama-upc.edu" –> "DAMA" i "UPC").

Pel que fa a la cerca de sigles en el nom, el primer que es farà és tokenitzar el nom a partir dels seus espais. Amb el nom tokenitzat, es consideraran sigles:

- La concatenació del primer caràcter de cada una de les paraules. ("Universitat Politècnica Catalunya" –> "UPC").
- Una paraula que estigui entre parèntesis ("Universitat Politècnica de Catalunya (UPC)" –> "UPC").
- La concatenació del primer caràcter de cada una de les paraules que estiguin entre parèntesis ("Universitat Politècnica de Catalunya (Facultat Informàtica Barcelona)"–>"FIB").
- Una paraula tal que el seu segon i quart caràcter siguin punts ("U.P.C." –> "UPC").
- La concatenació del primer caràcter de cada una de les paraules del nom separades per guions ("Universitat Politecnica Catalunya - Facultat Informàtica Barcelona" –> "UPC" i "FIB").
- La concatenació del primer caràcter de cada una de les paraules del nom separades per barres ("Universitat Politecnica Catalunya / Facultat Informàtica Barcelona" –> "UPC" i "FIB").

Cal tenir en compte que a l'hora de realitzar la concatenació de caràcters no es consideraran aquelles paraules que pertanyin a una llista de paraules que es volen evitar i que s'obtenen a partir d'un fitxer. Així, no es consideraran articles o preposicions que podrien portar a obtenir sigles incorrectes. D'aquesta manera, "Universitat Politècnica Catalunya" i "Universitat Politècnica de Catalunya", tindran la mateixa sigla: "UPC".

En el cas que al fer la concatenació de caràcters només hi hagi una sola paraula, aquesta paraula serà considerada una possible sigla. A més, si una sigla només té 2 caràcters, s'afegirà com a possible acrònim la mateixa sigla invertida (("Barcelona University" –> "BU" i "UB").

Després de tot aquest procés, s'obtindrà un fitxer amb totes les parelles "identificador d'institució - sigla" que s'hagin trobat.

5.5.3 Càrrega de sigles en una base de dades

Per tal de treballar d'una manera més còmoda en la deduplicació d'institucions es realitzarà una càrrega de totes les dades referents a institucions i sigles en una base de dades

temporal, tal com es fa en la deduplicació de ciutats. En aquesta base de dades, que al utilitzar *Dex* com en la resta del projecte tindrà forma de graf, hi haurà les institucions i els acrònims com a nodes, a més a més d'arestes que relacionin cada una de les institucions amb totes les seves possibles sigles.

La informació referent als acrònims provindrà d'un fitxer d'acrònims únics generat a partir del fitxer de relacions entre institucions i sigles, que serà l'usat per carregar les relacions en la base de dades. La informació referent a les institucions serà extreta del fitxer de participacions amb el nom normalitzat, la qual cosa permetrà realitzar una deduplicació d'institucions per acrònims molt similar a la realitzada anteriorment per noms. Així doncs, s'obtindrà una base de dades amb l'esquema de la Figura 5.12 .

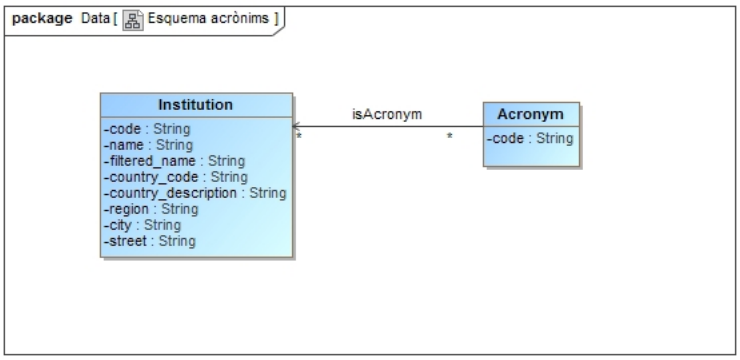


Figura 5.12: Esquema de la base de dades d'acrònims.

5.5.4 Creació de blocs

Input

Al Block Provider de *Dexdup*, arribaran sis elements a partir dels quals realitzar els blocs:

- Fitxer de ciutats úniques.
- Fitxer de països únics.
- Fitxer de tokens a eliminar de les adreces.

- Base de dades amb les ciutats.
- Base de dades amb els acrònims.
- Base de dades temporal inicialitzada.

Com es pot observar, els quatre primers elements coincideixen amb els utilitzats en la deduplicació d'institucions per nom. A més, també s'utilitzarà la base de dades amb tota la informació referent a sigles i institucions, i una nova base de dades temporal que tal i com es veurà més endavant servirà per evitar comparacions ja realitzades.

Implementació funcions *hasNext()* i *next()*

Per fer aquesta deduplicació es torna a utilitzar Dexdup. Igual que en la deduplicació per noms, serà necessari obtenir els blocs a partir de la interfície *BlockIterator,* que té dues funcions: *hasNext()* i *next().*

La primera vegada en que s'executa *hasNext(),* es realitza una inicialització. Els fitxers de ciutats úniques, països i tokens a eliminar es carreguen a memòria i s'inicialitzen els lectors. A més, també s'inicialitzaran els valors necessaris per poder utilitzar les diferents bases de dades i es crearà una llista amb tots els acrònims. Aquesta funció retornarà *true* mentre el bloc d'institucions actual tingui més de 2 elements o hi hagi acrònims a la llista que encara no s'han tractat.

D'altra banda, la funció *next()* farà el següent: si l'últim bloc retornat tenia dos o menys elements es crearà un nou bloc. Per crear aquest nou bloc s'agafarà el següent acrònim de la llista, i es buscarà en la base de dades totes les institucions que estan relacionades amb aquest acrònim. A continuació, s'obtindrà la informació de cada una d'aquestes institucions del graf i s'inserirà en el bloc. La dades de les institucions seran les mateixes que en el cas de la deduplicació per noms. Així, s'obtindran les dades de ciutat, país, regió...però també aquelles per les que es realitzen alguns processos intermedis com el nom normalitzat, les alternatives de la ciutat o l'adreça normalitzada. A més també s'afegirà l'acrònim per a aquesta deduplicació.

En l'Algorisme 5.19 es mostra quin serà el funcionament de la funció *hasNext()* del BlockIterator amb més detall, mentre que en l'Algorisme 5.20 es mostrarà el codi de la funció next() Cal destacar que la idea és agafar un bloc d'institucions que poden tenir

un mateix acrònim i comparar-les totes amb totes. Com Dexdup encara no ho permet, s'ha d'emular aquest comportament amb la funció *hasNext*(), que va retornant true si en un grup d'institucions que comparteixen un mateix acrònim encara no s'han comparat totes contra totes.

Algorisme 5.19 Pseudocodi del BlockIterator de la deduplicació d'acrònims

```
1  boolean init = false;
2  private Values acronyms = graph.getValues(acronym_code_attr);
3  boolean endvalue = true;
4
5  function hasNext(): boolean {
6      if (!init) {
7          inicialitzar(); //inicialitza readers, writers, identificadors d'elements
               del graf d'acrònims...
8      }
9
10     if (!endvalue || acronyms.hasnext()){
11         return true;
12     }
13     else {
14         acronyms.close();
15     }
16 }
```

Algorisme 5.20 Pseudocodi del BlockIterator de la deduplicació d'acrònims

```
1  ArrayList<Object> block = new ArrayList<Object>();
2
3  function next(): ArrayList<Object> {
4
5      //Acronym sobre el que es forma el block
6      String acronym = "";
7
8      if (endValue) {
9          //S'hagafa un nou valor sobre el que iterar i omplir el block
10         block.clear();
11         Objects o = graph.select(acronym_code_attr, Graph.OPERATION_EQ, acronyms.
               next());
12         long oidAcronim = o.first();
13         acronym = graph.getAttribute(oid, acronym_code_attr).toString();
14
15         //S'obtenen les institucions que tenen aquell acrònim
16         Objects institutionsOids = graph.neighbors(oidAcronim, isAcronym_type,
               Graph.EDGES_BOTH);
17
18         while (institutionsOids.size()<2 && acronyms.hasNext()) {
19             //Si el acrònim no té 2 o més institucions, es busca un altre fins a
                   trobar-lo o fins acabar els acrònims
20             o.close();
21             institutionsOids.close();
22             o = graph.select(acronym_code_attr, Graph.OPERATION_EQ, acronyms.next
                   ());
23             oid = o.first();
24             acronym = graph.getAttribute(oid, acronym_code_attr).toString();
25             institutionsOids = graph.neighbors(oid, isAcronym_type, Graph.
                   EDGES_BOTH);
26         }
27
28         //Si no hi ha més acrònims, es tanquen els objectes oberts i es retorna un
                block buit.
29         if (!acronyms.hasNext()) {
30             o.close();
31             institutionsOids.close();
32             return block;
33         }
34
35         //S'afegeixen les dades al block per a cada institució
36         for (long instOid: institutionsOid){
37             Institution inst = createInstitution(instOid); //crea una institució
                   amb les dades de instOid
38             block.add(inst);
39         }
40
41     }
42     else {
43         if (block.size() >= 3) {
44             block.remove(0);
45         }
46     }
47     if ((block.size()) <= 2) {
48         endValue = true;
49     }
50     return block;
51 }
```

5.5.5 Comparador

En aquest cas, la classe que implementarà el comparador serà *UsingComparablesAcronym*. Aquesta classe serà molt similar a l'utilitzada en el cas de la deduplicació per nom, però s'introduiran uns petits canvis referents a les sigles.

Com s'ha vist, el càlcul final de la similitud entre dos institucions està determinat per cinc valors que calculen la similitud de noms, ciutats, països, regions i adreces.

Alguns d'aquests valors, requereixen càlculs previs, com en el cas del valor de la comparació de noms. Aquest valor serà el màxim de diferents comparacions. Si en la anterior deduplicació es tenia en compte les comparacions amb el nom normalitzat, amb el nom sense espais o les comparacions per tokens, ara s'afegirà una nova comparació per acrònims. En la figura 5.13és pot veure el diagrama que descriu el procés del càlcul de similitud d'institucions per acrònim.

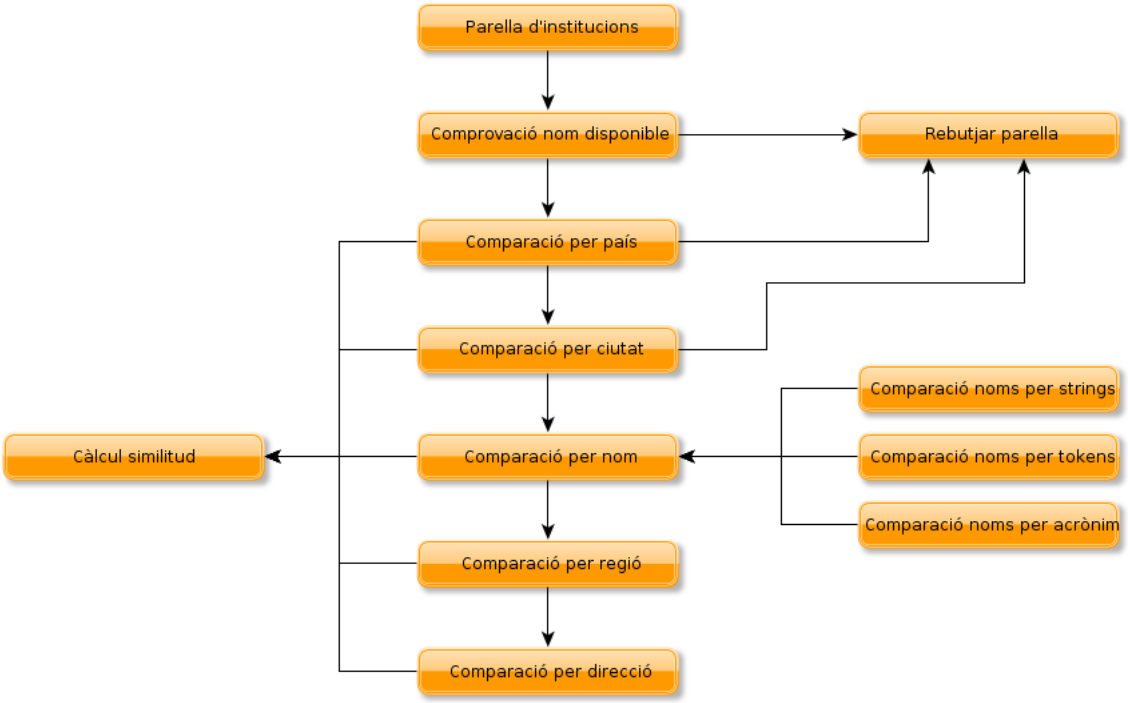


Figura 5.13: Procés de comparació d'institucions per acrònim.

Per obtenir el valor de la comparació per sigles, es mirarà si l'acrònim d'una institució és igual al nom de l'altre institució. Si es dóna aquest cas, s'establirà un valor d'entre 0.9 i 1 en funció de la longitud de l'acrònim (a major longitud de l'acrònim, més fiabilitat tindrà).

Per exemple, si tenim les següents institucions:

- *inst1*:
  - Nom: Universitat Politecnica de Catalunya.
  - Acrònim: UPC.
- *inst2*:
  - Nom: UPC.
  - Acrònim: UPC.

El resultat de la comparació per noms serà molt baixa, ja que "Universitat Politècnica de Catalunya" i "UPC" són molt diferents. Tot i així, com l'acrònim de *inst1* és igual al nom de *inst2*, el valor de la comparació per acrònims estarà entre 0.9 i 1. En aquest cas concret, al tenir la sigla tres caràcters, el valor obtingut seria 0.95.

D'altra banda, utilitzar aquest valor com a resultat de la similitud entre noms en casos on no hi hagi més dades (degut a que camps com el país o la regió son nuls) podria portar a obtenir falsos positius en la deduplicació. És per això, que per tenir en compte aquest valor de la comparació de noms per acrònims només s'utilitzarà en el cas que hi hagi com a mínim dos camps més no nuls que es puguin comparar.

A més, al tenir una incertesa més gran amb el valor de la comparació per noms, se li donarà un pes menor, passant de 0.7 a 0.6. Aquest percentatge restant anirà pels valors de les comparacions dels camps de ciutat, regió i país, que tindran un pes conjunt de 0.4 si no existeix la direcció o de 0.34 en cas contrari.

Per exemple, si tenim els següents valors:

- *compNom* (abans de la comparació per acrònim): 0.2.

- compAcronim:0.95
- *compCiutat*: 1.
- *compPais*: Nul.
- *compRegio*: 0.7.
- *compDireccio*: 0.8.

El càlcul per obtenir el resultat seria:

$$Max(0.2, 0.95) * 0.6 + 1 * (0.34/2) + 0.7 * (0.34/2) + 0.8 * 0.06 = \\ 0.57 + 0.17 + 0.119 + 0.048 = 0.907$$

5.5.6 Resultats

La classe que implementa el resultat en la deduplicació d'institucions per acrònims serà exactament la mateixa que en la deduplicació per noms, ja que l'esquema dels resultats és idèntic i només s'han realitzat modificacions en els càlculs de la comparació.

Els fitxers resultants seran els següents:

- Fitxer amb totes comparacions que estan per sobre del valor llindar: *duplicats-confirmats-acronym.csv*.
- Fitxer opcional (segons l'opció *WriteRejected*) amb totes comparacions realitzades que estan per sota del valor llindar: *duplicats-rebutjats-acronym.csv*.
- Fitxer amb comparacions entre institucions amb noms diferents però camps molt semblants: *duplicats-nom-acronym.csv*.
- Fitxer amb comparacions entre institucions amb ciutats nules però per sobre del llindar: *cityNulls-acronym.csv*.

5.5.7 Fitxer de configuració

El fitxer de configuració de *Dexdup* utitlitzat en el procés de deduplicació d'institucions per acrònim serà el descrit a continuació:

```
<config>

  <deduplication name="acronym" active="true">

    <blockProvider bean="BlockIteratorEqualNodeValues">
      <property name="cityFile" value="city-uniq.csv"/>
      <property name="tokensFile" value="token-address-list.csv"/>
      <property name="countryFile" value="countries.csv"/>
      <property name="separator" value="|"/>
      <property name="quoteChar" value="\0"/>
    </blockProvider>

    <comparisonMethod name="LevenshteinDistance" bean="UsingComparablesAcronym">
      <property name="cutRejected" value="true"/>
      <property name="nullWeight" value="0.5"/>
      <property name="minPosibleNull" value="3"/>
      <property name="threshold" value="0.9"/
      <propertyComparator name="Levenshtein" comparator="
          LevenshteinDistanceComparable"/>
      <propertyComparator name="LetterPair" comparator="LetterPairComparable"/>
      <propertyComparator name="EqualComparator" comparator="EqualComparable">
        <property name="nullWeight" value="0.3"/>
      </propertyComparator>
    </comparisonMethod>

    <result bean="ResultInstitutionPrinter">
      <property name="outputFile" value="duplicats-confirmats-acronym.csv"/>
      <property name="rejectedFile" value="duplicats-rebutjats-acronym.csv"/>
      <property name="cityNullsFile" value="cityNulls-acronym.csv"/>
      <property name="dupNameFile" value="duplicats-nom-acronym.csv"/>
      <property name="separator" value="|"/>
      <property name="quoteChar" value="\0"/>
      <property name="threshold" value="0.9"/>
      <property name="writeRejected" value="false"/>
    </result>

  </deduplication>

</config>
```

5.6 Creació de grups de duplicats

Arribats a aquest punt, es disposa de sis fitxers amb parelles d'institucions que es poden considerar la mateixa. Tot i això, no es vol obtenir una llista de parelles duplicades, sinó que es vol aconseguir obtenir grups de duplicats. És a dir, si tenim que A i B són duplicats, i que B i C també ho són, volem obtenir un grup de duplicats on hi hagi A,B i C, encara que no s'hagi realitzat la comparació entre A i C. Això es pot veure més clarament a la Figura 5.14. En aquesta, es mostra com hi ha dos grups d'elements semblants, que al estar units per un mateix element que és similar als dos conjunts, fa que es creï un grup de duplicats que conté tots els elements dels dos grups.

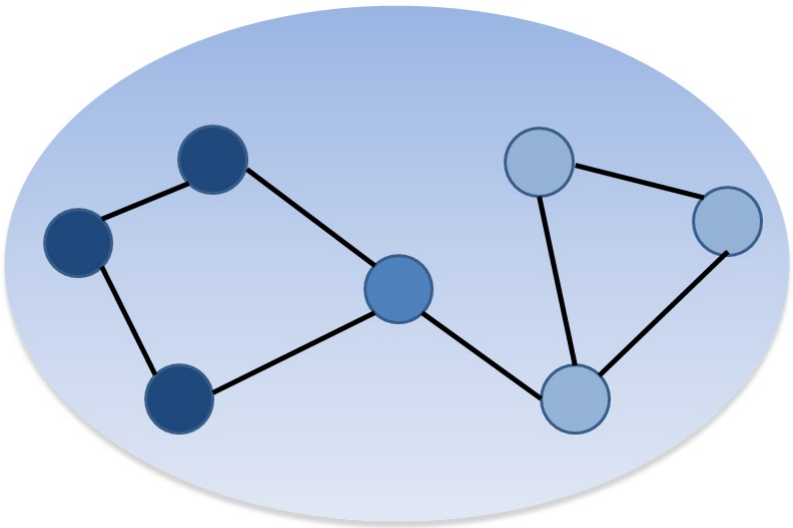


Figura 5.14: Exemple de grup de duplicats.

El primer pas a realitzar serà unir els fitxers del mateix tipus, per quedar-nos només amb tres: *duplicats-confirmats.csv, duplicats-nom.csv* i *cityNulls.csv*, que posteriorment serà ordenat per identificador d'institució.

Per tal de crear aquests grups de duplicats, es començarà per crear tres maps amb informació:

- Un map on hi haurà, per a cada institució, algunes de les dades utilitzades en la comparació que utilitzarem per decidir quina institució es la representant dins d'un grup: *hashInfo*.

- Un map indicant, per a una institució concreta, en quin grup està: *hashVisited*.

- Un map indican,t per a un grup, quines institucions té: *hashGroups*.

A continuació, s'anirà llegint el fitxer de *duplicats-confirmats*. Per a cada entrada, s'afegirà la informació de les dues institucions al map d'informació *hashInfo* sempre que no hi hagi aquesta informació en el map. A més, es mirarà si els identificador de les institucions estan en el map *hashVisited*.

- Si cap dels identificadors ha estat inserit en el map, s'insereixen indicant que estaran en un mateix grup que porta per identificador l'identificador de la primera de les dues institucions. A més, es crearà aquest grup al map *hashGroups*.

- Si un identificador no està en el map però l'altre si, s'insereix el primer indicant que estarà en el grup en el que està el segon. També s'afegirà en la llista d'identificadors d'institucions d'aquest grup l'identificador de la institució que encara no estava inserida.

- Si els dos identificadors han estat inserits en el map, s'unifiquen els grups als que pertanyien cada una de les institucions. Serà necessari ajuntar les llistes d'institucions de cada grup en un i eliminar-ne l'altre. A més serà necessari actualitzar el map *hashVisited* per a cada una de les entrades que es passen del segon grup al primer.

Una vegada s'hagi realitzat aquesta part del procés, que es mostra en l'Algorisme 5.21, es disposarà d'un map amb tots els grups de duplicats, i per cada grup, una llista amb les institucions que hi pertanyen.

El següent pas és tractar el fitxer de duplicats d'institucions amb ciutats nul·les. Amb el sistema utilitzat, aquests duplicats presenten un problema important: al no tenir ciutat, pot ser que s'hagi determinat en el proces de deduplicació que una institució A que no té ciutat sigui un duplicat de dos institucions B i C de ciutats diferents. Això provocaria que els grups de les institucions B i C s'ajuntessin, cosa que no es desitja.

Algorisme 5.21 Pseudocodi de la creació de grups

```
function createGroups(): void {
    File dupFile = duplicats-confirmats.csv
    HashMap hashVisited = new HashMap<Long, Long>();
    HashMap hashGroups = new HashMap<Long, List<Long>>();
    CSVReader reader = new CSVReader(dupFile);

    String[] line;

    while ((line = reader.readNext())!= null){
        long id1 = Long.parseLong(line[0]);
        long id2 = Long.parseLong(line[1]);

        if (hashVisited.containsKey(id1)) {
            long ref = hashVisited.get(id1);
            if (hashVisited.containsKey(id2)) {
                if (hashVisited.get(id2) != ref) {
                    // uneix els dos grups als que pertanyen id1 i id2, deixant-ne
                        només un
                    unifyGroups(hashVisited.get(id2), ref);
                }
            }
            else{
                hashVisited.put(id1, (long) ref);
                hashVisited.put(id2, (long) ref);
                List<Long> list = hashGroups.get((long) ref);
                list.add(id2);
                hashGroups.put((long) ref, list);
            }
        }
        else {

            if (hashVisited.containsKey(id2)) {
                long ref = hashVisited.get(id2);
                hashVisited.put(id1, (long) ref);
                List<Long> list = hashGroups.get((long) ref);
                list.add(id1);
                hashGroups.put((long) ref, list);
            }
            else {
                hashVisited.put(id1, id1);
                hashVisited.put(id2, id1);
                List<Long> list = new ArrayList<Long>();
                list.add(id2);
                hashGroups.put(id1, list);
            }

        }
    }
}
```

Per solucionar-ho, s'ha decidit que una institució que no tingui ciutat només es pugui afegir en un grup, però no pugui provocar la unificació de dos grups diferents. La següent qüestió es determinar amb quin grup s'ha d'unir.

La millor solució trobada consisteix en buscar totes les parelles de duplicats de la institució sense ciutat, i comprovar quina d'aquestes institucions és la que té una similitud més elevada per tal d'unir la institució sense ciutat amb aquesta. En el cas que hi hagi varies institucions amb una mateixa similitud, s'unirà la institució sense ciutat amb la institució que pertanyi a un grup amb més duplicats.

Finalment, queden per tractar les parelles de duplicats que s'assemblen molt en totes les dades excepte en el nom. El que es farà en aquests casos és mirar a quin grup de duplicats pertanyen i amb quins grups s'ajuntarien. Si tots els elements del grup al que pertanyen tenen aquesta mateixa relació (ser molt similar en totes les dades excepte en el nom) amb com a mínim una institució de l'altre grup, els grups s'ajuntaran. Un altre cop, es realitza d'aquesta manera per evitar que s'uneixin grups de duplicats que en realitat no ho són.

Una vegada finalitzat aquest procés, existiran grups d'institucions duplicades, però serà necessari tenir aquesta informació en forma de parelles de duplicats. Per això, s'aniran recorrent els diferents grups i s'aniran escrivint en un fitxer les parelles formades per l'identificador de la primera institució afegida al grup (que servirà com a identificador de grup) i tota la resta d'institucions.

5.7 Actualització d'identificadors d'institucions

En aquest punt, es disposarà del fitxer de participacions i d'un fitxer amb parelles de duplicats en el qual tots els elements d'un mateix grup de duplicats tenen un mateix identificador. Aquest identificador serà el de la institució al qual s'assemblen. Així doncs, el que es farà és substituir l'identificador existent en el fitxer d'institucions pel del fitxer de parelles de duplicats.

Això es realitzarà posant prèviament tota la informació de les parelles de duplicats en un map, per tal de, posteriorment, anar recorrent el fitxer d'institucions substituint els identificadors corresponents. Aquest procés, així com el de l'apartat 5.9, seguiran el mateix algorisme presentat anteriorment en l'apartat 4.2.6.

5.8 Selecció de representant de grup

Després d'actualitzar els identificadors de les institucions, en el fitxer de participacions hi haurà varies institucions amb un mateix identificador. D'entre totes aquestes, se n'haurà d'escollir una, que serà la representant del grup de duplicats. Seleccionar una bona institució per fer de representant és important, ja que quan es facin peticions per aquesta institució, les dades de ciutat, país, pàgina web... seran les d'aquesta. Així doncs, s'haurà de seleccionar com a representant aquella institució d'un grup que tingui unes dades més completes.

El primer pas serà ordenar el fitxer de participacions segons l'identificador de l'institució. A partir d'aquí, per a cada grup d'institucions duplicades s'elegirà aquella que tingui més camps vàlids. A més, en cas que fos necessari, es canviaria el valor del camp ciutat pel valor més repetit en els duplicats (cal recordar que una mateixa ciutat pot estar escrita de diferents maneres). Aquest procés es realitzarà per mitjà de diferents maps, que contindran informació sobre les vegades que es repeteix un nom de ciutat o el nombre de valors no nuls que té una institució. Així, al recórrer el fitxer de participacions ordenats s'aniran actualitzant aquests maps per tal de després poder escriure el nou fitxer directament recorrent els maps.

En el fitxer resultant hi haurà el llistat d'institucions úniques, format per les institucions que s'han seleccionat com a representats de cada grup de duplicats.

5.9 Actualització de referències

De la mateixa manera que es realitza en el fitxer de participacions, també hi ha altres fitxers que necessitaran actualitzar els identificadors de les institucions. Concretament, els fitxers que necessitaran modificacions seran el de coordinadors únics i el de relacions entre acrònim i institució. Per fer-ho, simplement serà necessari substituir els identificadors a partir del fitxer de parelles de duplicats segons els grups.

Tot i així, en alguns casos, aquesta substitució pot portar a tenir relacions repetides que s'hauran d'eliminar. D'aquesta manera, en els casos del fitxer de participacions i del fitxer amb la relació entre acrònim i institució, es realitzarà una comprovació per tal d'eliminar totes aquelles parelles "identificador de projecte - identificador d'institució" i

“acrònim - identificador d’institució) que ja s’hagin trobat anteriorment en el fitxer i per tant siguin repetides.

Un cop realitzat això, els fitxers ja estaran preparats per ser inserits en la base de dades, de manera que es passaria a la fase de càrrega de dades, ja comentada en el capítol anterior.

6 Experiments

6.1 Conceptes previs

Una de les coses importants a tenir en compte a l'hora de fer un projecte d'aquest tipus és la necessitat de realitzar algun tipus de validació dels resultats obtinguts després de realitzar la deduplicació. Fer aquest procés és especialment important al tractar amb grans volums de dades, ja que encertar en la deduplicació de casos concrets no garanteix que sempre s'encerti. Així doncs, en els casos de deduplicacions, hi ha dos conceptes importants a tenir en compte a l'hora de validar resultats: la precisió i el *recall*. Per obtenir els valors d'aquesta precisió i del *recall*, s'han realitzat dos experiments que es comentaran més endavant.

6.1.1 Precisió

Entenem per precisió la qualitat de les nostres prediccions de similitud. En concret, estaríem parlant del nombre de deduplicacions correctes entre totes les que s'han trobat. D'aquesta manera, per calcular la nostra precisió, serà necessari obtenir un conjunt de dades de prova que serà un conjunt de les deduplicacions trobades i observar quantes d'aquestes deduplicacions efectivament són correctes.

6.1.2 Recall

D'altra banda, el *recall* és la quantiat de deduplicacions que s'obtenen amb el nostre sistema d'entre totes les existents. Així doncs, per calcular el *recall* s'agafarà un subconjunt de les dades originals, i es mirarà quantes de les deduplicacions d'aquest conjunt s'han obtingut efectivament amb el sistema utilitzat.

Cal tenir en compte que la precisió i el *recall* són inversament proporcionals, ja que com més alt sigui el valor del recall (s'obtè un percentatge més alt de les deduplicacions existents), més fàcil serà que alguna d'aquestes deduplicacions sigui incorrecta, disminuint la precisió. Cal recordar que una de les fites importants a l'hora de realitzar aquest projecte era obtenir una bona precisió, encara que això afectes negativament en el *recall*, ja que a l'inici del projecte es va considerar més crític ajuntar dues institucions diferents que una mateixa institució estigués separada en dues instàncies diferents.

6.1.3 Interval de confiança - Bernoulli

A l'hora de validar els resultats obtinguts i donar un nombre concret per la precisió i el *recall*, és necessari tenir en compte un altre concepte com és l'interval de confiança.

En l'estadística, rep el nom d'interval de confiança a una parella de nombres entre els quals s'estima que estarà un cert valor desconegut amb una determinada probabilitat d'encert. Aquests nombres que determinen un interval es calculen a partir d'unes dades de mostra, i la probabilitat d'èxit en la estimació, que rep el nom de nivell de confiança, es representa amb$1-\alpha$. En aquest cas, α és l'anomenat error aleatori o nivell de significació, que és una mesura de les possibilitats d'equivocar-se en la predicció.

El nivell de confiança i l'amplitud del interval varien de forma conjunta, de manera que un interval més ampli tindrà una major possibilitat d'encert (un nivell de confiança més elevat), mentre que un interval més petit tindrà una possibilitat d'error més elevada, tot i que també s'obtindrà una estimació més precisa.

Així doncs, per calcular l'interval de confiança de les estimacions de precisió i *recall*, s'utilitzarà la següent fórmula:

$$Intervaldeconfiança = \hat{p} \pm z_{1-\frac{1}{2}\alpha} \sqrt[2]{\frac{1}{n}\hat{p}(1-\hat{p})}$$

En aquesta fórmula, $\hat{p}$ és el resultat de la successió d'encerts en els processos de Bernoulli. Els processos de Bernoulli són un conjunt d'experiments aleatoris on només pot haver-hi dos resultats: encert o error. De forma menys tècnica, és el que s'ha explicat anteriorment al parlar de la precisió i el rec*all*.

D'altre banda, n és el nombre d'elements que té la mostra amb la que s'ha fet els experiments, i α és l'error aleatori. Per a obtenir un valor amb un nivell de confiança

estàndard en aquest tipus de proves del 95%, α serà un 5%. Això fa que en aquest cas, $z_{1-\frac{1}{2}\alpha} = 1.96$.

6.2 Experiment 1 - Precisió

Per aquest primer experiment, s'ha buscat obtenir un valor per a la precisió mostrada pel sistema de deduplicació creat a l'hora de deduplicar les institucions de CORDIS. Per obtenir aquesta precisió és necessària la creació d'una mostra de les deduplicacions trobades pel nostre sistema. Per fer aquesta mostra, s'han contemplat dues possibilitats diferents. Per una banda, és podria agafar un nombre determinat de parelles que s'hagin considerat com a duplicats i fer l'experiment a partir d'aquesta mostra, però no seria del tot correcte. Això és degut a què una part molt important del sistema, la creació de grups de duplicats (on institucions que directament no s'han considerat com a duplicats poden acabar pertanyent al mateix grup), no es tindria en compte. És per això que s'ha decidit agafar com a mostra un subconjunt dels grups de duplicats trobats al final de tot el procés.

Tot i això molts d'aquests grups estan formats per una sola institució (amb el que no hi ha cap duplicat en aquests grups). D'altra banda, trobem grups amb un gran nombre d'institucions, de manera que comprovar manualment si totes les relacions dins d'un grup són correctes seria un procés molt laboriós. D'aquesta manera, el que s'ha fet és obtenir 50 grups aleatòriament entre tots aquells grups que tinguessin més de 10 i menys de 50 institucions. Aquests 50 grups, contenen en total gairebé 1000 instàncies de institucions, de manera que ja és una mostra d'un tamany força considerable. Per tenir una referència millor, en general hi ha 105179 grups (nombre de institucions úniques obtingudes després de tot el procés), mentre que de grups que compleixin les condicions anteriors n'hi ha 3412.

D'aquests 50 grups seleccionats aleatòriament, només en un cas s'ha trobat una institució que no pertanyia en aquell grup. Així doncs, seguint la fórmula de l'apartat anterior tenim els següents paràmetres:

- $\hat{p} = \left(1 - \frac{1}{50}\right) = 0.98$
- $n = 50$

- $z_{1-\frac{1}{2}\alpha} = 1.96$

D'aquesta manera, el valor de la precisió serà el següent:

$$\hat{p} \pm z_{1-\frac{1}{2}\alpha} \sqrt[2]{\frac{1}{n}\hat{p}(1-\hat{p})} = 0.98 \pm 1.96 \sqrt[2]{\frac{1}{50}0.\hat{9}8(1-0.98)} = 0.98 \pm 0.0388$$

6.3 Experiment 2 - Recall

En el segon experiment s'ha buscat obtenir el valor del *recall* del sistema de deduplicació. Trobar aquest valor és una mica més complex que en el cas de la precisió, ja que trobar una petita mostra aleatòria d'institucions en les que hi hagi duplicats és complicat, i augmentar la mida de la mostra complica força la cerca manual de duplicats.

Així doncs, el que s'ha buscat és obtenir un subconjunt d'institucions on potencialment hi hagi duplicats mitjançant un filtre per tal de fer aquest experiment. En concret, s'han obtingut totes les institucions que tenien en el camp ciutat una de les cinc ciutats catalanes següents: Girona, Lleida, Tarragona, Igualada i Terrassa. El conjunt de totes aquestes institucions crea una mostra de 761 institucions.

Segons el sistema de duplicació, aquestes 761 institucions es divideixen en 123 institucions úniques. Fent la revisió de duplicats manual, s'han trobat 5 institucions úniques que haurien de pertànyer a un altre grup. D'aquesta manera, segons la mateixa fórmula que s'ha aplicat per la precisió obtenim amb els següents paràmetres:

- $\hat{p} = \left(1 - \frac{5}{123}\right) = 0.9593$
- $n = 123$
- $z_{1-\frac{1}{2}\alpha} = 1.96$

D'aquesta manera, el valor del *recall* serà el següent:

$$\hat{p} \pm z_{1-\frac{1}{2}\alpha} \sqrt[2]{\frac{1}{n}\hat{p}(1-\hat{p})} = 0.9593 \pm 1.96 \sqrt[2]{\frac{1}{50}0.9593(1-0.9593)} = 0.9593 \pm 0.0178$$

D'altra banda, addicionalment també s'ha calculat la precisió d'aquesta mostra de dades. En aquest cas, s'han trobat dos institucions col·locades en dos grups incorrectes. Així doncs, els valors dels paràmetres serien els següents:

- $\hat{p} = \left(1 - \frac{2}{123}\right) = 0.9837$
- $n = 123$
- $z_{1-\frac{1}{2}\alpha} = 1.96$

Per tant, el resultat de la precisió és:

$$\hat{p} \pm z_{1-\frac{1}{2}\alpha} \sqrt[2]{\frac{1}{n}\hat{p}(1-\hat{p})} = 0.9837 \pm 1.96 \sqrt[2]{\frac{1}{50}0.9837(1-0.9837)} = 0.9837 \pm 0.0174$$

Els resultats de cada ciutat es poden veure en la taula de la taula 6.1, on els falsos positius són el nombre d'institucions que s'han detectat com a duplicats i no ho són, i els falsos negatius són aquelles institucions que no s'han detectat com a duplicats però que si que ho són en realitat.

Ciutat	**Institucions**	**Grups duplicats**	**Falsos positius**	**Falsos negatius**
Igualada	69	20	1	0
Lleida	127	16	0	2
Tarragona	314	30	1	1
Girona	123	26	0	2
Terrassa	128	31	0	0
TOTALS	**761**	**123**	**2**	**5**

Taula 6.1: Resultats de l'experiment 2.

6.4 Anàlisi dels resultats

Un cop realitzats els experiments és moment de revisar els resultats amb detall. La primera cosa que s'ha de tenir en compte, és que en tots els experiments s'ha decidit treballar a nivell de grup (és a dir, d'institucions úniques finals) en lloc de a nivell d'institucions inicials. S'ha decidit fer així perquè es representa una mica millor el que realment es vol aconseguir amb el projecte, encara que fent els experiments a nivell d'institucions inicials s'obtenen valors de precisió i *recall* sensiblement més elevats.

Pel que fa a la precisió, es pot observar que el valor calculat de l'experiment 2 (0.9837 $\pm$ 0.0174) coincideix en l'interval del experiment 1 (0.98 $\pm$ 0.0388), sempre parlant amb

una confiança del 95% que és l'estàndard en aquests casos. Tot i això, encara que en serveix per tenir una idea força aproximada de la precisió del sistema de deduplicació, aquest valor tampoc és exacte degut a la forma en què s'ha format el conjunt de mostra. Per una banda, hi ha molts grups amb una o dues institucions que farien augmentar molt la precisió, mentre que per contra, també hi ha alguns grups amb centenars d'entitats en els quals seria més probable que hi hagués alguna institució situada incorrectament.

Tot i això, queda ben clar que una de les fites d'aquest projecte es compleix, ja que en qualsevol cas s'obté una precisió molt elevada.

Pel que fa al *recall*, també hi ha un aspecte que fa que el resultat obtingut no sigui del tot exacte. Una part important del sistema de deduplicació són els *mètodes de blocking*, que indiquen quines institucions es comparen amb quines. En aquest experiment, aquests *mètodes de blocking* no hi queden reflectits, ja que al ser la mostra de l'experiment relativament petita en comparació amb el conjunt sencer de dades, es compararan les institucions totes contra totes. Així doncs, aquest resultat del *recall* real podria ser lleugerament més baix.

Tot i així, els valors obtinguts són molt positius, ja que tot i tenir una gran precisió, el valor de *recall* obtingut es manté molt alt.

7 Planificació i costos

Després d'haver descrit d'una forma exhaustiva quin és el sistema utilitzat per a dur a terme la deduplicació de les dades de CORDIS, és moment de presentar la planificació i la viabilitat econòmica en les quals caldrà recolzar-se per a poder dur a terme el desenvolupament del projecte. El fet de tenir organitzades les tasques d'un procés és realment important en l'àmbit de l'enginyeria, ja que permet implementar una aplicació en els límits temporals i econòmics establerts per un client.

7.1 Planificació

La planificació d'un projecte no ha de ser estàtica, és a dir, no ha de ser una tasca que es realitzi en un moment concret i que ja no es torni a modificar, sinó que comença el primer dia en què sorgeix un projecte i s'ha d'anar canviant i adaptant fins al lliurament final del projecte segons les necessitats que vagin sorgint. Una bona planificació ha de descriure l'organització que es durà a terme, l'assignació de recursos i la duració de les tasques a desenvolupar durant el temps establert per a realitzar un projecte. En un inici, aquesta planificació és una previsió del temps que serà necessari per a cada tasca a realitzar, i a partir d'aquesta planificació es podrà obtenir un pressupost, que és el que marcarà la viabilitat econòmica del projecte un cop presentat al client.

Aquesta planificació inicial es realitza en base a l'experiència adquirida en altres projectes similars ja acabats, la qual cosa permetrà predir quin temps serà necessari per a completar cada tasca amb força exactitud. Tal i com ja s'ha dit, aquesta planificació es veurà modificada degut a les inevitables desviacions que s'aniran produint en el transcurs d'un projecte. Així doncs, aquesta planificació acaba resultant una descripció real de les hores que s'han anat destinant a cada una de les feines a realitzar, la qual cosa permet calcular els costos reals del projecte.

En aquest capítol es mostrarà únicament la versió final de la planificació feta per a aquest projecte, és a dir, la planificació que és imatge de la feina real feta. Així doncs, no es presentaran les versions inicials i intermèdies per tal de simplificar l'explicació d'aquest capítol. Aquesta planificació final es presentarà mitjançant un Diagrama de Gantt, que permet representar de forma visual el calendari d'un projecte i els terminis i dependències existents entre les diferents tasques que el componen. Aquestes tasques se subdividiran en altres tasques més petites i concretes per a poder arribar al detall de com s'ha fet aquest projecte en el temps. A més de les tasques pròpies del projecte, també s'han tingut en compte les diferents dates importants per a la realització del PFC. Aquestes dates inclouen la matriculació o la presentació de l'informe previ.

Per a la realització d'aquest Diagrama de Gantt i el càlcul de costos, s'ha utilitzat l'eina de gestió de projectes anomenada Microsoft Project 2013. El resultat es mostra en les figures 7.1 i 7.2.

Figura 7.1: Diagrama de Gantt (1).

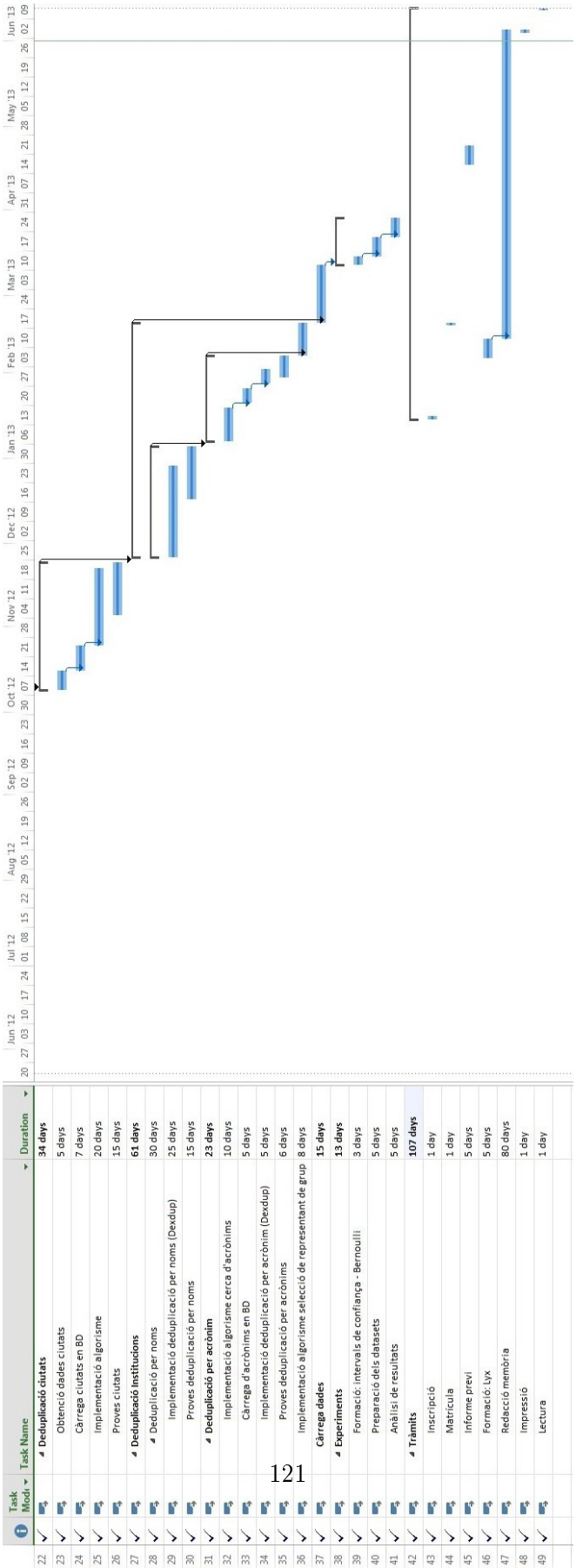


Figura 7.2: Diagrama de Gantt (2).

7.2 Costos

Com s'ha remarcat anteriorment, és tan important la realització prèvia d'una planificació com la modificació d'aquesta segons els desviaments que s'hagin produït durant el transcurs del projecte. La primera ens serveix per poder determinar un pressupost, que servirà per poder comprovar si hi ha els recursos necessaris per tal de realitzar totes les tasques en el temps determinat. D'altre banda, la versió final de la planificació ens ajudarà a calcular els costos finals del projecte i que, per tant, el client haurà de pagar.

En la majoria de projectes (i aquest no n'és cap excepció) hi intervenen dos tipus de recursos principals: els recursos humans i els materials. Els recursos humans són les persones que treballen en el projecte i a qui se'ls hi assignen les diferents tasques a realitzar, mentre que els recursos materials estan formats per totes aquelles eines que ajuden a la realització de les tasques d'un projecte. Així, un programador seria un recurs humà, mentre que l'ordinador que necessita per a fer la seva feina seria un recurs material. A més, en el cost total d'un projecte també s'haurien d'incloure les despeses per la realització de desplaçaments, per la utilització del sistema elèctric, l'amortització de les eines emprades durant el procés... Aquests altres recursos més difícilment mesurables no seran tinguts en compte en el càlcul dels costos.

7.2.1 Recursos humans

Les persones que treballen en el projecte, que anteriorment hem designat com a recursos humans, són els recursos que representen un volum més important del conjunt de despeses, tan en aquest com en la majoria de projectes. Tot i així, en aquest projecte totes les tasques han estat realitzades per un becari de projecte, amb l'ajuda de la figura del Director de projecte. Això provoca que no ha existit un grup de persones especialitzades en cada una de les tasques a realitzar. D'aquesta manera, per a fer més real el càlcul de costos final, s'ha cregut oportú realitzar una simulació d'un grup de recursos humans a qui se'ls hi assignaran les diverses tasques exposades en la secció anterior en el Diagrama de Gantt. No obstant, la quantitat de temps invertit en cada una de les tasques és totalment real.

Per a la nostre simulació, tindrem un conjunt de dues persones implicades en el projecte: un analista, i un programador. L'analista serà l'encarregat de fer l'anàlisi, especificació

i disseny de l'aplicació juntament amb tota la documentació necessària. El programador serà el que durà a terme les decisions preses per l'analista i, per tant, s'encarregarà de la codificació de l'aplicació. Els sous d'aquests treballadors ficticis es veuen reflectits en la figura 7.3.

Nom del recurs	**Tarifa estàndard**
Analista	35€/hora
Programador	25€/hora

Figura 7.3: Tarifes fixades pels treballadors ficticis.

En la planificació representada en el Diagrama de Gantt, s'estima que el 40% del temps dedicat al projecte serà per part de l'analista i que el 60% restant serà pel programador. D'aquesta manera, tenint en compte els sous i les hores treballades, obtenim els següents càlculs:

Per a als 276 dies laborals necessaris per a la realització d'aquest projecte, a 7 hores diàries de feina, s'obtenen 1932 hores dedicades al projecte. L'analista haurà fet el 40% d'aquestes hores, és a dir unes 772 hores. D'altra banda, el programmador haurà treballat les 1160 hores restants. Amb aquestes dades, tenim que s'hauran invertit 772 x 35 = 27.020 euros en el sou de l'analista i 1160 * 25 = 29.000 euros en el sou del programador. Així doncs, el cost dels recursos humans serà 27.020 + 29.000 = 56.020 euros.

7.2.2 Recursos materials

Els costos materials representen els costos referents a totes les eines utilitzades pel personal per al desenvolupament del software. Així, també és important afegir els derivats de la utilització de les màquines i del software que han estat necessaris per a arribar a finalitzar el projecte.

Per tal de ser coherents amb la simulació dels diferents tipus de recursos humans, s'ha calculat que hauran estat necessaris dos equips informàtics equipats amb els seus perifèrics corresponents. A més, també serà necessari una tercera màquina més potent on es puguin realitzar les càrregues més costoses. A més, també cal tenir en compte els

diferents costos associats a les llicències del software utilitzat. Aquests costos es poden veure en la Figura 7.4.

Nom del recurs	Cost
Equips informàtics	2 x 1.000€ = 2.000€
Equip potent	2.000€
Magic Draw Standard Edition	400€
Microsoft Windows 7	100€
Microsoft Office 2013	120€
Microsoft Project 2013	400€

Figura 7.4: Costos fixats pels diferents recursos materials.

Tot això, fa que el cost dels recursos materials s'elevi fins als 5.020€.

7.2.3 Costos totals

Així doncs, una vegada s'han calculat els costos relacionats amb el personal i amb el material utilitzat, podem fer el càlcul final del cost del projecte, que serà la suma d'aquests dos apartats. Cal recordar que aquests costos no són reals, ja que que es tracta d'un projecte de final de carrera, però que aquests costos estan realitzats a partir d'una simulació en base a una planificació de tasques i una estimació temporal totalment real.

Així doncs, tal i com es pot veure en la Figura 7.5, el cost total del projecte serà el cost dels recursos humans (56.020€) més el cost dels recursos materials (5.020€). Això fa que el projecte en total tingui un cost de 61.040 euros.

Tipus de cost	Valor
Recursos humans	56.020€
Recursos materials	5.020€
Total	61.040€

Figura 7.5: Cost total del projecte.

8 Conclusions i futur del projecte

Aquest últim capítol té com a objectiu fer una valoració de la feina realitzada i posar de manifest tot allò que m'ha aportat a nivell personal i professional.

8.1 Valoració projecte

Una vegada arribats en aquest punt, es pot dir que els objectius d'aquest projecte s'han assolit de manera molt satisfactòria. S'ha aconseguit deduplicar un conjunt de dades complex com el de Cordis amb una gran precisió sense sacrificar gaire en el aspecte del *recall*, tal i com s'ha mostrat en el capítol dels experiments. D'aquesta manera, s'ha pogut obtenir un conjunt de dades net a partir del qual poder utilitzar l'aplicació web de *Sciencea*.

A més, durant el procés de resoldre el problema amb els duplicats, s'ha creat un *framework* de deduplicació com *Dexdup*, aplicable no només a aquest projecte sinó a altres deduplicacions de caire molt més genèric.

A nivell personal, em sento molt satisfet de la feina feta durant tot el projecte, ja que s'ha treballat molt per tal d'arribar a obtenir aquests bons resultats finals. A més, el fet de realitzar el projecte dins de l'estructura d'un grup de recerca com DAMA-UPC m'ha possibilitat aprendre molt dels companys, cadascun en la seva àrea. Sempre han estat disposats en ajudar en tot moment i han resolt els diferents dubtes que han anat sorgint durant l'elaboració del projecte, la qual cosa valoro molt. A més, tot i que anteriorment no ho tenia gens clar, estar en un grup com aquest, tan a nivell humà com professional, m'ha servit per decidir-me a seguir treballant en aquest camp en el futur.

8.2 Coneixements previs i adquirits

En la realització d'aquest PFC he pogut posar en pràctica diferents coneixements adquirits durant la carrera, però també n'he adquirit altres de nous degut a la necessitat de trobar solucions als diferents problemes que han anat sorgint.

A part dels coneixements que es treballen a les assignatures de programació que es realitzen durant la carrera, han estat especialment importants els coneixements d'assignatures com *Enginyeria del Software 1*, *Enginyeria del Software 2* i *Bases de Dades*. A més, tampoc s'han de deixar de banda les coses apreses en altres assignatures com *Presa de Decisions i Gestió de Projectes Empresarials*, que m'han permès planificar i organitzar el projecte d'una manera força més eficient.

D'altra banda, el fet de realitzar aquest projecte també m'ha aportat nous coneixements adquirits, la majoria d'ells de caire pràctic, tal i com és habitual en un projecte final de carrera. Alguns d'aquests coneixements són els referents al *Record Linkage*, la implementació de software amb Java, la utilització de l'IDE d'*Eclipse* [15], l'utilització de l'eina de gestió i construcció de projectes *Maven* [16] o l'utilització de *Lyx* [17] per a la generació de documents basada en *Làtex*.

8.3 Treball futur

De cara al futur, s'ha de tenir en compte que *Sciencea* és una aplicació web que seguirà activa, de manera que poden requerir-se noves modificacions en les dades si alguna nova funcionalitat ho requereix. Un exemple d'això seria que es volguessin tenir en compte no només els projectes europeus sinó de tot el món. En aquest cas, s'hauria de buscar una font d'informació similar a CORDIS, extreure les dades, analitzar-les i trobar una manera d'unificar-ho tot.

Tot i així, si ens centrem en el projecte en sí, es podria treballar de cara a millorar-ne l'optimització, reduint així el temps que es tarda en realitzar tot el procés. Més concretament, hi ha dos parts del codi que podrien ser optimitzades:

- En la deduplicació d'institucions es repeteixen comparacions i posteriorment s'eliminen els resultats repetits. Això s'ha fet d'aquesta manera perquè no es podia

mantenir un registre a memòria de totes les comparacions realitzades (n'hi ha massa per poder garantir que no hi haurà problemes per falta de memòria) i tampoc es podien emmagatzemar aquestes comparacions en una base de dades temporal, ja que el temps de consulta és superior al temps de recalcular una comparació. En un futur, es podria estudiar la viabilitat d'utilitzar una base de dades externa a *Dex* per tal de guardar aquestes comparacions. Una opció podria ser *LevelDB*, que treballa en memòria però que a la vegada és persistent.

- Durant la càrrega es realitzen algunes ordenacions de fitxers que poden arribar a tenir una mida molt gran. Aquestes ordenacions, per evitar fer-les en memòria usen *Dex* i això provoca que el procés d'ordenació s'alenteixi. Una solució viable seria realitzar un nou algoritme d'ordenació basat en el *Sort* de *Linux*, on el fitxer a ordenar s'emmagatzema en parts dins de petits fitxers ordenats que després es tornen a ajuntar de manera ordenada utilitzant un algoritme *MergeSort*.

D'altra banda, en la part de les deduplicacions de les institucions o de ciutats es podria investigar per tal d'utilitzar altres camps d'informació que apareixen en les dades i que en un principi s'han descartat degut a que en poques ocasions hi ha informació vàlida. En concret, es podria treballar sobre els camps relatius a la posició geogràfica (latitud i longitud) d'una institució. Això permetria ajustar amb més precisió la ciutat, regió i país d'una institució que tingués en aquests camps algunes dades incorrectes o mancants. Tot i així, no seria una feina fàcil de resoldre.

Una altre possibilitat per tal d'ampliar el realitzat en aquest projecte podria ser donar més opcions al *framework* de *Dexdup*. Per exemple, es podria donar l'opció per comparar de diferents maneres els blocs retornats pel *BlockIterator*, de manera que es pogués triar si es vol utilitzar el sistema actual de comparacions per un grup (*one vs all*) o si es vol que es comparin tots els objectes amb tots. Aquesta última opció faria que la deduplicació de regions fós més eficient al no haver de fer consultes repetides a la base de dades.

9 Bibliografia

[1] Cordis, http://cordis.europa.eu/

[2] Sciencea, http://www.sciencea.com

[3] Dama-upc, http://www.dama.upc.edu.

[4] Do, H.-H., and Rahm, E. Coma: a system for exible combination of schema matching approaches. In VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases (2002), VLDB Endowment, pp. 610621.

[5] Michalowski, M., Thakkar, S., and Knoblock, C. Exploiting secondary sources for unsupervised record linkage. In Proceedings of the 2004 VLDB Workshop on Information Integration on the Web (2004).

[6] Rahm, E., and Bernstein, P. A. A survey of approaches to automatic schema matching. The VLDB Journal 10, 4 (2001), 334350.

[7] Deen, S. M., Amin, R. R., and Taylor, M. C. Data integration in distributed databases. IEEE Trans. Softw. Eng. 13, 7 (1987), 860864.

[8] Rohan Baxter, L. G. Decision models for record linkage. In Data Mining (2006), pp. 146160.

[9] Baxter, R., and Christen, P. [14] Cohen, F. A comparison of fast blocking methods for record linkage.

[10] Yan, S., Lee, D., Kan, M.-Y., and Giles, L. C. Adaptive sorted neighborhood methods for ecient record linkage. In JCDL '07: Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries (New York, NY, USA, 2007), ACM, pp. 185194.

[11] DEX, http://www.sparsity-technologies.com/dex.php

[12] Comissió Europea, http://ec.europa.eu

[13] Konstantinidis, S. Computing the Levenshtein distance of a regular language.IEEE Information Theory Workshop (2005). 8884884.

[14] Geonames, http://www.geonames.org

[15] Eclipse, http://www.eclipse.org/.

[16] Apache maven 2, http://maven.apache.org/.

[17] Lyx: The document processor based on tex/latex, http://www.lyx.org/.

[18] Eurostat, http://epp.eurostat.ec.europa.eu/portal/page/portal/nuts_nomenclature/introduction